Adaptive Visual Representations for Autonomous Mobile Robots using Competitive Learning Algorithms

A Thesis
Submitted to the
Faculty of Graduate Studies
in Partial Fulfillment
of the Requirements
for the Degree of
Doctor of Philosophy

Dean K. McNeill

Department of Electrical
& Computer Engineering
University of Manitoba

0-612-35045-2

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES
*****
COPYRIGHT PERMISSION PAGE

ADAPTIVE VISUAL REPRESENTATIONS FOR AUTONOMOUS MOBILE ROBOTS USING COMPETITIVE LEARNING ALGORITHMS

BY

DEAN K. McNEILL

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University

of Manitoba in partial fulfillment of the requirements of the degree

of

DOCTOR OF PHILOSOPHY

DEAN K. McNEILL ©1998

Abstract

This thesis examines issues surrounding the class of unsupervised artificial neural network learning algorithms known as competitive learning. Four variations of competitive learning algorithms are presented and compared, both theoretically and based on their relative performance in the solution of a number of low and high dimensional input environments. In particular, the thesis discusses efficacy of these algorithms in learning appropriate representations of visual information in robots. Comparisons of hard competitive learning (HCL) and soft competitive learning (SCL) in the low dimensional discrimination of Gaussian data clusters showed that SCL consistently produces superior solutions. As well, the tendency of HCL to become trapped in sub-optimal solutions was analysed and found to be an inherent shortcoming of the winner-take-all nature of the algorithm. It was also found that selection of an appropriate network size may be achieved through the use of a simple pruning technique if a surplus of network units are provided to begin training. Further investigations involving HCL, SCL, and both the DeSieno and Krishnamurthy implementations of frequency sensitive competitive learning (FSCL) show that the latter ($FSCL_K$) produces the most consistently reliable solutions to a number of learning tasks. This result was obtained as a consequence of extensive testing involving a high dimensional data clustering problem. That problem concerned the adaptive identification and classification of motion via an array of optical sensors residing on an autonomous mobile robot. The selection and arrangement of sensors used by this robot were derived from the vision system of jumping spiders. Operation of an integer-only version of $FSCL_K$ on the actual robotic hardware demonstrates the system's ability to cluster some aspects of the motion identification task. The inability to completely identify and generalize to novel input patterns is attributed to deficiencies in the sensors used and is not an inherent shortcoming of the algorithm. These deficiencies can be corrected through the use of some preprocessing of the raw sensor readings. As well, during the course of this study the winner-take-all activations used by the frequency sensitive algorithms were replaced with analog activations, resulting in significantly improved network generalization.

Acknowledgements

I would like to express profound thanks to my advisor-in-all-things, Howard Card, for his boundless scientific curiosity, wealth of inspirations, and great friendship, all of which were invaluable in the completion of this thesis.

As well, I'd like to express my appreciation to the examining committee for the considerable time and effort spent reading and evaluating this work.

I would also like to thank the many other friends and colleagues who have made my time at the University of Manitoba a most enjoyable and rewarding experience.

Lastly, and most certainly not least, I would like to thank my family for their support and encouragement in all my endeavours, academic and otherwise.

Financial support for this work was received through the Natural Sciences and Engineering Research Council of Canada (NSERC), and from Micronet, a Network of Centres of Excellence. Support in the form of computing resources was also provided by the Canadian Microelectronics Corporation.

Table of Contents

List of Figures

List of Tables

Introduction

1

Artificial Neural Networks (ANNs) have been employed to significant advantage in a variety of signal and information processing applications. They are particularly useful in situations where a straightforward deterministic algorithm for the mapping of system inputs to outputs is not available, or where such a mapping is too complex to be coded effectively using traditional techniques. Examples of such applications include handwritten character and digit recognition,[1,2] real-time navigation of autonomous vehicles,[3] weapons detection in airline baggage,[4,5] sonar target identification,[6,7,8] malignant cell recognition,[5] and prediction of stock market trends.[5] These problems are difficult to solve because one can not easily identify the underlying properties of the problem space which are required for the formulation of an analytical solution. However, it is generally easy to provide examples that are representative of the task required. Paradoxically, these types of tasks are frequently the kinds which humans possess a natural ability to solve, in spite of our inability to describe the mechanism leading to that solution. The strength of the ANN approach stems from the network's ability to evolve an appropriate output response based solely on the available examples (training data) and a simple learning rule. It accomplishes this by extracting relevant high-level characteristics from the training data, which are then used to formulate the output responses. What is particularly powerful about this parametric representation is the network's ability to apply this acquired higher-level knowledge to input data which it has not previously encountered during the training process. This property, which is known as the network's ability to *generalize*, results in a very robust system, which is a critical requirement for the successful solution of the above problems.

While there are a growing variety of neural network algorithms to choose from, they can all be separated into two major categories; supervised learning and unsupervised learning. In the case of supervised learning, the data used to train the network consists not only of the input data (x), but also an associated desired network response or target output (t).[1] Through the learning rule the network adjusts its free parameters, the network weights ($\mathbf{w}$), in an attempt to produce the desired input-response association. Once training is complete the neural system is theoretically able to emulate the behaviour of the actual system which originally generated the outputs (t) from the inputs (x). Figure 1 gives a block diagram representation of this arrangement. The true measure of the quality of the resulting neural model is how faithfully it predicts the response of the original system for values of the inputs which were not part of the training data. If the ANN performs well in this regard it may then be used as a reasonable substitute for the original system, such as a human being. One example of where such a network has been used very successfully is in the automated sorting of mail by reading the handwritten postal codes on envelopes.[1] The Backpropagation learning algorithm[9] was used in this application and is the most common and widely studied example of supervised learning.

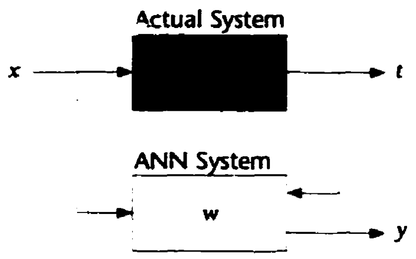


Figure 1: Generative model of a neural network.

1. Vector quantities are printed in bold type, while scalar quantities are printed in plain type.

While supervised algorithms are quite powerful, they can only be used in situations where one is able to assemble a set of training data showing the desired input-target associations. Yet, in many cases it is not known in advance what these associations should be, or frequently even how many network nodes are required to best represent the data. In spite of these handicaps there exists an underlying structure to the data which, if discovered, can be used to formulate a model of the system that produced it. Unsupervised learning algorithms are capable of doing exactly that. They attempt to evolve a high-level representation by identifying global statistical properties of the data distribution. These properties generally manifest themselves as clusters or groupings of data points in the input space and as a result the algorithms are sometimes referred to as clustering algorithms. Once identified, these higher level features may be used directly or may serve as inputs to additional processing stages. This thesis examines a particular class of unsupervised learning algorithms known as *competitive learning*. The task of these algorithms is to compress a complex, possibly high dimensional, sensory space into a simpler representation consisting of a limited set of system states.

The central problem explored in this thesis is the unsupervised learning of efficient representations of visual environments encountered by simple mobile robots. The results are also believed to be relevant to other portable embedded computing applications. A variety of competitive learning algorithms are investigated and compared for their suitability in this task. Several modifications to some of these algorithms were made in order to improve their performance or reduce their computational complexity. However, the primary contribution of this thesis was not in improving these algorithms but in evaluating their usefulness in these applications. This process was guided by reference to the vision system of a simple animal, in an effort to exploit the extensive experimentation which has already been performed by biological evolution. The vision system of a jumping spider was chosen for its appropriate input dimensionality, quality of visual sensors, and estimated computational capacity.

It is generally believed that most of the computational burden in vision is associated with properly representing the data, rather than classifying it for behavioural responses. The raw visual data obtained via the robot's sensory apparatus is an impoverished representation of the visual scene. A multitude of raw sensory vectors actually correspond to the same environmental situation, with their differences resulting from simple shifts in the image, time delays, or intensity variations for example. The task of the unsupervised learning is to cluster similar situations into the same system state in the new feature space. These learned states of the system are known as *visual representations*, and are a prerequisite to any subsequent supervised learning. Based on this dimensionally reduced representation the mobile robot then performs a variety of stereotyped behaviours in association with each state.

As a result of their importance to the vision task, adaptive visual representations obtained without supervision using artificial neural learning became the central focus of this thesis.

1.1 Competitive Learning

As the name would suggest, competitive learning (CL) networks drive the learning process by employing some form of competition between the various units within the network. There are a number of variations of competitive learning with the most notable being standard or hard competitive learning (HCL), soft competitive learning (SCL), frequency sensitive competitive learning (FSCL), and self organizing feature maps (SOFM). While all of these methods are closely related, there are some important differences which affect the way in which these algorithms extract information from the input data and how that information is used to guide the learning process.

1.1.1 Standard or Hard Competitive Learning

The most basic type of competitive learning is standard or hard competitive learning.[10,8] It attempts to motivate the learning process by enforcing strict competition between units in the network. Figure 2 illustrates the typical structure of an HCL network. The actual training of the network is a relatively straightforward process. For each input data pattern (x), each unit computes the distance (h_i) between that pattern and the unit's weight vector (w_i). A variety of distance metrics may be used here, including the Euclidean distance, Minkowski metric, and

Mahalanobis distance.[14] For the purpose of this discussion we will use the simple Euclidean distance as given in Equation 1.

$$h_i = \| \mathbf{w}_i - \mathbf{x} \| \tag{1}$$

The unit with the smallest h_i is designated as the winning unit (i^*) for that particular input vector and sets its output (activation) high (1). All the other units which lost the competition set their activations low (either 0 or −1). The winning unit then updates its weights in order to reinforce the association between itself and the input pattern. This is accomplished by adjusting the weight vector by a small amount in the direction of the input pattern.

$$\Delta w_{i^*,j} = \varepsilon(x_j - w_{i^*,j}) \tag{2}$$

Since only the winning unit performs a weight update this learning method is sometimes known as *winner-take-all* (WTA) learning. The symbol ε in equation 2 is known as the learning rate and controls the magnitude of the weight adjustment. It is fixed at a small value prior to the start of training and usually remains unchanged through the entire learning procedure. The selection of ε can have a significant impact on the overall success of the learning process. If ε is too large the network will make large weight adjustments for each input pattern. Since the objective is to extract global features, this places too much emphasis on any one pattern. Instead it is desirable that only small weight changes be made, thus permitting the overall trend in the data to emerge. A lesser problem is selecting too small a learning rate, resulting

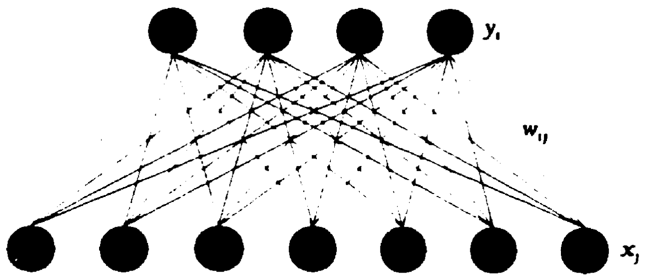


Figure 2: Typical structure of a competitive learning network.

in very small weight changes and long learning times. A value on the order of $\varepsilon=0.001$ is typically used.

If one considers this entire process geometrically with the weights normalized to unit length such that

$$\sum_i w_{ij}^2 = 1 \qquad (3)$$

and the inputs normalized in a similar fashion, then both the inputs and outputs can be represented as points on a unit hypersphere. The winning unit in any competition will be the unit whose weight vector is closest to the current input vector. Under these conditions minimization of the Euclidean distance is equivalent to maximizing the inner product $\mathbf{w}_i \cdot \mathbf{x}$.

The greatest advantage of the HCL algorithm is its simplicity. It does, however, have serious limitations under certain circumstances. It is common practice to begin the training process by first initializing the network weights to random values. In some situations this randomization will result in a unit being positioned well outside the bounds of the input data distribution. Figure 3 illustrates such an initial distribution in a two dimensional data space. Clearly the unit at (0.5,0.5) is closer to all of the

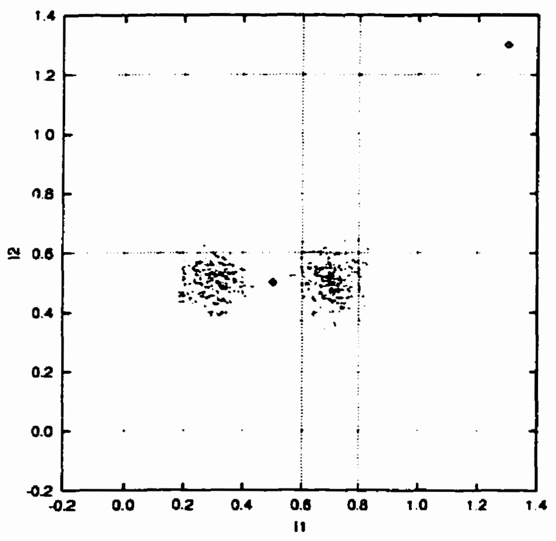


Figure 3: Possible data distribution showing complex clusters and two weight vectors.

data points and, as a result, will win the competition for every point. Since only the winning unit updates its weights the unit at (1.3,1.3) will never make a weight adjustment. It will, therefore, never become a participating member of the network's solution and is effectively orphaned. As a consequence the network will fail to learn any useful discrimination of the input patterns. Fortunately, several alternatives exist which can address the orphaned unit problem.

1.1.2 Frequency Sensitive Competitive Learning

Frequency Sensitive Competitive Learning (FSCL) attempts to correct HCL's shortcomings by introducing a conscience mechanism[11] into the competition. This mechanism has the effect of reducing the likelihood that a unit will win subsequent competitions each time it wins the current competition. This eliminates the situation where only one unit wins for a large majority of the training cases. Under this scheme the unit closest to a particular data point will win initially, but eventually, due to the conscience penalty, the unit on the periphery of the data will later win and be moved closer to the data. Eventually it will be drawn close enough to the data that it will win a significant percentage of the time and will then contribute in a reasonable manner to the solution.

The FSCL algorithm itself is similar to HCL in its initial stages. As with HCL the FSCL network first computes the Euclidean distance between the input and the weight vectors and determines the winning unit. The activation of this unit is again set high, and all others low.

$$\begin{array}{c} y_i = 1 \text{ if } \|w_i - x\| < \|w_j - x\| \forall j \neq i \\ y_i = 0 \text{ otherwise} \end{array} \tag{4}$$

However, unlike HCL the winning unit may not necessarily be the unit which performs a weight update. The network first computes a bias (b_i) for all units using the equation

$$b_i = C\left(\frac{1}{N} - p_i\right) \tag{5}$$

Here N is the number of units participating in the competition, and C is a bias factor which determines the distance a losing unit must reach in order to enter the solution. The value p_i is a measure of the fraction of time the unit i wins a competition, and can be computed using the following equation.

$$p_i^{new} = p_i^{old} + B(y_i - p_i^{old}) \tag{6}$$

where $0 < B << 1$. The value of B determines how sensitive the conscience mechanism is to the winning of a single competition. As was the case with the learning rate, a small B value should be selected in order to ensure that the overall statistics of the data distribution are used to drive the competition. B should not be so large that any single vector of the dataset unduly influences the competition. As a result of his investigations, DeSieno[11] recommends a B value of 0.0001.

The bias value b_i is now used in the calculation of a new winning unit such that

$$\begin{aligned} z_i &= 1 \text{ if } (\|w_i - x\|^2 - b_i) \le (\|w_j - x\|^2 - b_j) \forall j \ne i \\ z_i &= 0 \text{ otherwise} \end{aligned} \tag{7}$$

Having thus determined the winner under the influence of the conscience penalty, the weights of that unit are updated according to equation 2. It should be noted that the value z_i is only used in determining which unit's weights are to be updated. The output activation of the units is still determined by equation 4.

A related variant of FSCL has been proposed by Krishnamurthy et al.[32,33] It is a slightly simpler method which introduces the frequency dependence into the distance computation by mean of a fairness function $F(u_i)$.

$$h_i = F(u_i)\|w_i - x\| \tag{8}$$

The fairness function is a non-decreasing function of u_i, where u_i is a count of the number of times unit i has succeeded in winning a competition. A typical choice for this function is $F(u_i) = u_i$. As before, the unit having the smallest distance h_i is selected as the winner i* and its weights are then updated according to equation 2.

Unfortunately, the property of FSCL which makes it useful in helping to ensure the participation of all units is precisely the mechanism which introduces an additional problem. Use of the algorithm will result in the units adjusting their weights such that each unit wins an approximately equal proportion of the competitions. However, one can easily visualize situations where some of the data clusters are more densely populated than others, which should ideally result in some units winning more often in those clusters than in others. By requiring all units to win an approximately equal number of competitions the network is discouraged from assigning a single unit to a dense cluster, which would be the most appropriate solution. So while FSCL is able to achieve better performance than HCL, it does not always produce the best possible solution.

1.1.3 Soft Competitive Learning

An algorithm which corrects for HCL's orphaned unit problem, and also avoids the uniform frequency restriction of FSCL, is soft competitive learning (SCL). SCL is another variation on the basic HCL network, but with this algorithm there is no single winner associated with a particular input vector. Instead all units compute an activation based on their distance from the current input pattern, and learning corresponds to an on-line version of the Expectation Maximization (EM) algorithm.[34] Radial basis function (RBF) units[12,8] are commonly used in SCL networks instead of the linear units used in HCL and FSCL. Here the activation function of the RBF unit is a normalized Gaussian and is calculated using the following equation:

$$y_i = \frac{e^{-\|\mathbf{w}_i - x\|^2/2\sigma_i^2}}{\sum_i e^{-\|\mathbf{w}_i - x\|^2/2\sigma_i^2}} \tag{9}$$

As seen previously, x represents the input pattern vector and **w** the network weights. Here the weight vector **w** identifies the mean of the Gaussian activation function and the parameter σ_i controls the variance or spread of the function. Both **w** and σ_i are typically determined by the learning process. In the case of a symmetric Gaussian, σ_i is a diagonal matrix which can be decomposed to $\sigma_i = c_i\mathbf{I}$, where **I** is the identity matrix, and c_i are constants. Use of symmetric Gaussians is not strictly required and in many cases a non-symmetric Gaussian may provide a better fit to the data. In those situations σ becomes a full covariance matrix. This will obviously increase learning time, since it introduces additional degrees of freedom which the network must now explore in searching for a solution. A possible alternative to using the full covariance matrix is to provide additional symmetric units to the network, thus allowing it to cover an elliptical data cluster with several symmetric Gaussians, instead of a single non-symmetric function.

Special attention should be drawn to the fact that with SCL the activations are now analog quantities. These analog outputs represent the degree of partial membership which the input vector $\boldsymbol{x}$ has within the receptive fields of each unit. Or alternatively, it identifies the extent to which each unit is responsible for the data point $\boldsymbol{x}$. This technique of encoding the input vector as an aggregation of the network's analog activations is known as a *distributed representation*. Thus the result of the SCL algorithm is to find the optimal distributed representation of the input. This is in contrast to standard competitive learning where a *local representation* is obtained in which only a single unit is activated at one time for a given input vector. The weight update equation for SCL is similar to that given for HCL in equation 2, with the exception that all units perform weight updates in proportion to their activation.

$$\Delta w_{ij} = \varepsilon y_i(x_j - w_{ij}) \tag{10}$$

As a consequence of this procedure, every unit learns for every input pattern but to varying degrees in relation to that unit's activation. Geometrically the units whose weight vectors are closest to the input vector make the greatest move towards the input, while those farther away move only slightly closer. This is opposite to the weight updating used by HCL and FSCL (equation 2) which makes larger weight updates when the winning units are farther from the input point.

Under the SCL scheme, no single unit can end up in the situation of monopolizing the inputs at the expense of all other units. Every unit will eventually participate in the solution even if its weights were initialized well outside the data distribution.

This process may take quite a long training time in some cases, but all the units are guaranteed to eventually be used and not orphaned.

1.1.4 Kohonen Self-Organizing Feature Maps

A fourth algorithm, which is related to CL, is the self-organizing feature map (SOFM) developed by Kohonen.[13,14] This algorithm is one of the first and probably the most well known example of unsupervised learning. It has been widely studied in the literature and an updated examination of the area has recently been made by Kohonen.[14] As a result, the present work will not investigate the properties of this algorithm in depth. However, the fact that SOFMs share some similarity to HCL and SCL make them worth mentioning briefly.

The three CL algorithms discussed above place no special significance on the ordering of the units. They are only interested in identifying the features within the data and are unconcerned with which units ultimately represent which features. SOFMs, in contrast, attempt to evolve a topological representation of the input data in an unsupervised manner. The units themselves are typically arranged in a two dimensional map, though other arrangements are possible. The learning process begins by once again determining the unit with the smallest Euclidean distance between its weight vector and the input vector, and this unit is selected as the winner. However, unlike HCL and FSCL where only one unit is updated, the SOFM updates both the winner and all those units in a local *neighbourhood* of it. As learning progresses the neighbourhood slowly shrinks until, in the end, only the winning unit is being updated. This procedure ensures that all units achieve some degree of adaptation, with the selection of the neighbourhood function and its decay rate being critical factors in ensuring that all units participate in the solution. The SOFM process allows for entire regions of the map to become initially tuned to particular inputs and this tuning is then gradually refined as the neighbourhood shrinks. In the final solution logically adjacent inputs will activate neighbouring outputs, thus indicating a topological correlation.

1.1.5 Objective Functions

In all learning algorithms there exists an objective function which drives the learning. In the case of HCL, $FSCL_D$, and $FSCL_K$ the objective is the minimization of the squared-error between the weight vectors and a data vectors. This expression is given in equation 11, where *j* varies over the number of units in the network layer (*N*).

$$\text{Error} = \sum_j \|x - \mathbf{w}_j\|^2 \tag{11}$$

This error is summed over all inputs (*x*) for which the unit is considered the winner. Thus the lower the total distortion between the inputs and weight vectors, the better the performance of the network.

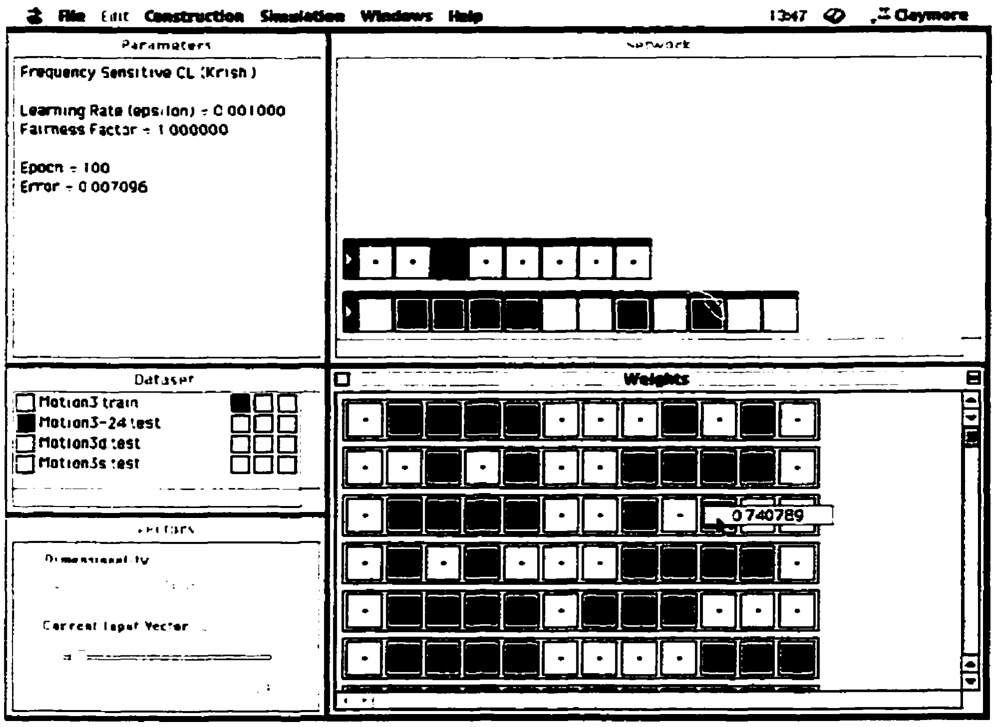
The objective function used by SCL is quite different than the other algorithms owing to the probabilistic nature of the RBF units. This network strives to maximize the probability that the units are responsible for generating the input data values. As will be demonstrated in chapter 3, this may result in markedly different placement of the weight vectors. The error measure used here is given in equation 12. Here *k* varies over the number of patterns in the training set (*P*), and *N* is the number of units in a layer.

$$\text{Error} = -\sum_k \ln\left(\frac{\sum_i e^{-\|\mathbf{w}_i - x\|^2/2\sigma_i^2}}{NP(2\pi\sigma^2)^{N/2}}\right) \tag{12}$$

Artificial Neural Network Simulator 2

Software simulation is the primary method used in the study of neural network algorithms and their applications. The preliminary experiments conducted in this thesis were performed using the Xerion artificial neural network simulator developed by the Artificial Intelligence Laboratory at the University of Toronto. That simulator was produced in the early 1990s primarily for that group's own research activities, but it was also made available to the general neural network community. Due mainly to the high amount of computation required in simulating complicated algorithms, this software was only available for the UNIX computing environment. As a consequence of the evolution and upgrading of the UNIX system used in our laboratory, Xerion became nonfunctional during the early stages of this study. At that same time, support and development of the simulator were discontinued at the University of Toronto.

As a replacement for Xerion, this author has written a new, custom designed, neural simulator and it is that simulator which was used to obtain the results reported in this thesis. This simulator, which we have named *Claymore*, was implemented in C on a Macintosh® and provides a full graphical user interface for convenience of visualization and ease of operation. In addition to the HCL and SCL algorithms which were available in Xerion, the new simulator also includes an implementation of both the DeSieno and Krishnamurthy versions of FSCL.



Frequency Sensitive CL (Krish)

Learning Rate (epsilon) = 0.001000
Fairness Factor = 1.000000

Epoch = 100
Error = 0.007096

Motion3 train
Motion3-24 test
Motion3d test
Motion3s test

0.740789

Figure 4: Screen capture of the Claymore ANN Simulator.

Figure 4 shows the graphical user interface for the Claymore simulator. The upper right window displays Hinton diagrams of the unit activations. These diagrams represent the magnitude of the activations by the size of the shaded area and the sign by its colour. Positive activations are shown in blue, while negative values are shown in red (not visible in this figure). In the case of this figure, a twelve input, eight output network is being simulated. The lower right window displays the corresponding weight vectors, again using Hinton diagrams. This graphical representation allows the user to easily observe the status of the network while training by visually monitoring changes to the weights. The user also has the option in both of these windows of selecting any element of the diagrams in order to view the actual floating point activation or weight value of that element. As can be seen in the figure, the particular weight element selected has a value of 0.740789.

The windows on the left provide feedback about the state of training and the selection of training set and test pattern. The upper left window is a status window and displays any algorithm specific parameters, such as learning rate. In addition, it shows the number of epochs simulated, and the network error resulting from that training.

The centre left window is the dataset window. It allows the user to select which datasets are to be used for training the network and which are for testing. The black selection indicator to the left of the dataset name permits the user to choose which of the datasets to use for network testing. The black selection indicator to the right of the dataset name, permits the user to select which dataset to use for training. Finally, the bottom left window is the test vector selection window. It permits the user to easily cycle through the test dataset (selected in the dataset window above) and to observe the resulting output activations in the network window. Patterns can be presented sequentially by using the arrow buttons, or an arbitrary pattern can be chosen with the slider control. Some basic attributes of the dataset are also shown at the top of this window.

Network construction and simulation controls are available through the corresponding menus. These provide options to add and connect layers, modify the algorithm specific parameters, randomize network weights, and perform training. Weight values can be saved and restored at any time through the File menu. As well, the loading of dataset files is also available under that menu.

2.1 Structure and Design Considerations of the Claymore Simulator

The overall design considerations for this software were extensibility and ease of use. The various program elements were organized to ensure that all user interaction with the program was provided solely through the graphical user interface. No external configuration files are required to set-up or simulate a network, thus making the operation of the simulator very intuitive.

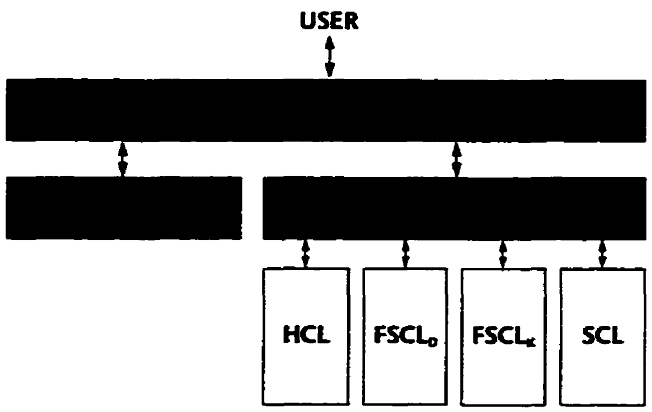


Figure 5: Organizational structure of the Claymore ANN simulator.

Just as important as the user interface was the requirement that the simulator be constructed in such a way as to allow for easy addition of new algorithms. Figure 5 shows a block diagram representation of the simulator. As can be seen from that figure, all interactions with the user are performed through the GUI. It then passes all simulation related requests to the simulation control module (SCM) which provides all general algorithm independent simulation functions. When it becomes necessary for an algorithm dependent operation to be performed the SCM calls the corresponding interface function in the algorithm module. Each of the algorithm modules is constructed in the same way and provide the same basic operations to the SCM. The diagram of figure 6 illustrates the interaction between these two modules. Any algorithm specific data structures for the network, layers, neurons, or synapses are allocated and initialized by the algorithm module in response to calls from the SCM (which performs initialization of the common parameters such as learning rate). Training and testing of the network is performed by calling the interface functions *Do Epoch* and *Apply Vector*. These then in turn call any other functions internal to the module in order to accomplish the request, such as updating weights and activations.

Strict adherence to this programming model allows new algorithms to be incorporated very quickly. Algorithm models of average complexity can be included

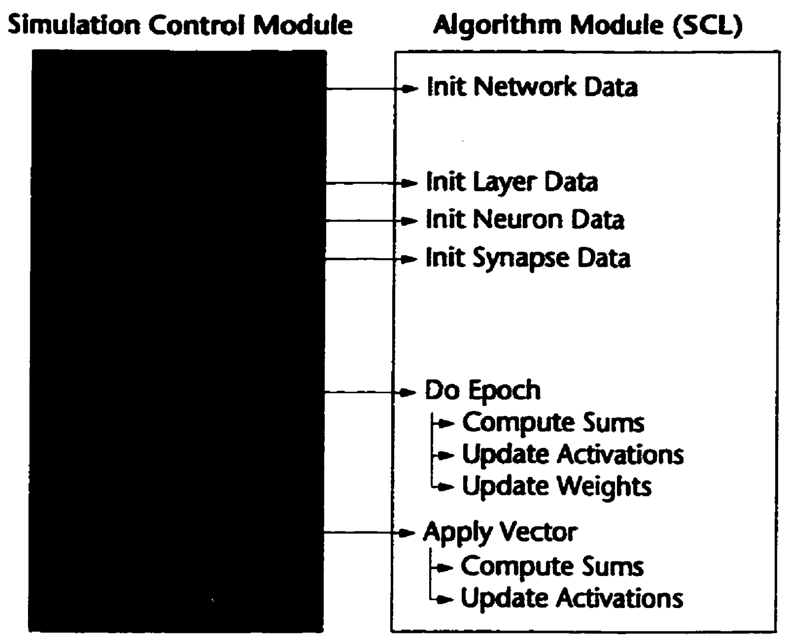


Figure 6: Interface between Simulation Control Module and Algorithm Modules.

in only a few hours. Source code for the four competitive learning algorithms programmed for this thesis are provided in Appendix A.

Empirical Examination of CL

3

The four variations of competitive learning introduced in chapter 1 are not new algorithms. They have all existed in the neural network literature for some time. However there has yet to be a detailed investigation conducted into the relative performance of these algorithms. The most complete examination reported to date was conducted by Krishnamurthy et al.[33] This involved a comparison of their implementation of FSCL to both HCL and SOFM. The work presented in this and subsequent chapters will extend this investigation to include SCL and the DeSieno variation of FSCL. It is hoped that the results of this analysis will provide a clearer understanding of the relative operation and performance of competitive learning algorithms. That knowledge will permit the intelligent application of competitive learning by providing us with a better appreciation for the class of problems which these algorithms are capable of solving and under what conditions deficiencies in the algorithms may impede their operation. With that in mind, we will first examine the performance of the algorithms on abstract low-dimensional problems, such as those which might be experienced by the robotic system of figure 7. This robot was the first constructed during this thesis and employed two analog optical sensors (or eyes) corresponding to a two dimensional input environment. In later chapters these investigations will be expanded to encompass a more complicated hardware based application operating in higher dimensional input spaces.

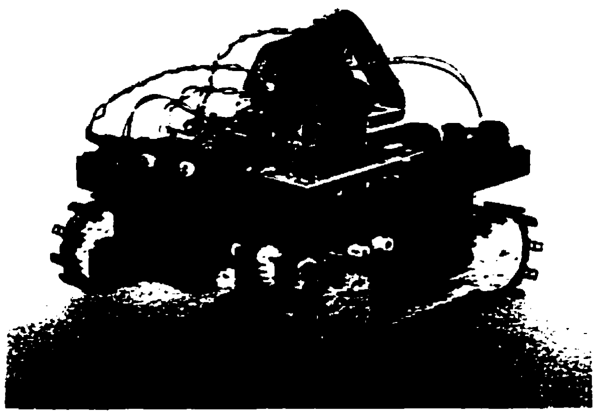


Figure 7: Mobile robot with two dimensional visual input.

3.1 Parametric Simulations of Competitive Learning

The investigation began with the development of a number of idealized test cases which permit examination of the algorithms under carefully controlled conditions and in situations where the solutions may be easily visualized and evaluated. These cases will be used to determine not only whether the neural algorithm is capable of reaching an acceptable solution, but also how quickly it converges to that solution and what difficulties it encounters along the way.

The test cases themselves consist of Gaussian distributions of data points in a two dimensional input space. In terms of the robot, these Gaussians correspond to similar visual scenes which provide comparable optical intensities measured by the two detectors. Since the test data has been artificially generated it possesses well known properties, allowing for precise evaluation of the network's performance. The following description will detail the results of simulations conducted on the datasets using the HCL and SCL algorithms.

The investigation first considered the case of simple geometrical data distributions which are well separated in the input space (corresponding to clusters of similar visual scenes, well separated from other distinct scenes). Of interest is the way the receptive fields (prototypes) orient themselves in relation to the data they are attempting to represent. First we consider the very simple case of two physically disjoint Gaussian clusters placed at opposite sides of the input space. A network with two output units was then simulated using these input patterns and it was found that both the SCL and HCL networks were able to learn this problem from a limited number of data points such that one unit was centred in each of the two input clusters. Figure 8 shows the input distribution and the learned location of the prototype centres (**w**). The SCL network used symmetrical Gaussian activation functions and for the purpose of these simulations the variance of these Gaussians was fixed at $\sigma^2=0.0044$ to match the spread of the data clusters, though this parameter could also be learned. The networks were trained using a dataset of 1000 patterns drawn from the two gaussian distributions with each gaussian producing half of the patterns. These input vectors were stored in the training file in random order to avoid any potential systemic effects which may arise from a sequential ordering. The network weights were initially set to random values also drawn from a Gaussian distribution centred at 0.5 with a standard deviation of 0.1. The results from this experiment showed that HCL performed slightly better on this problem than SCL, reaching a stable solution in only 4 passes through the training data

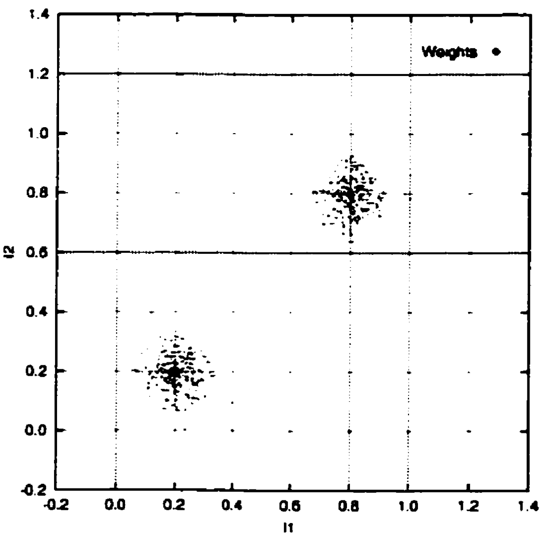


Figure 8: Two well separated Gaussian input clusters and the learned weight vectors.

(epochs). SCL reached an initial solution in 5 epochs and then refined that solution for an additional 5 epochs. A learning rate of $\varepsilon = 0.001$ was used by both networks.

For this experiment the weights began in the centre of the input space and moved quickly to the centres of the data clusters. However, in some situations the weights may be initially positioned in a region well away from the data values. If this occurs, a different learning behaviour is observed. With the HCL algorithm whichever of the two prototypes is closest to the input data will win the competitions and move towards them, taking up a position in the centre of all the data points. The other unit, being much farther away from the data, will lose every competition and the learning will end up stuck in a non-optimal solution. The second unit is orphaned and its presence is essentially irrelevant. With the SCL network, one initially notices a similar type of behaviour, but because all units update their weights in proportion to their activations the second unit still makes very small movements towards the data. Following several epochs it will eventually be drawn into the middle of the distribution. Once this occurs the units then diverge to cover the two separate data clusters as before. This result clearly shows an advantage to using the SCL method, though the number of epochs required to reach the final solution will be very large. For the simulations conducted, it was not uncommon for SCL to require 18000 epochs to converge. This long training time is a direct consequence of the very small weight adjustments dictated by equation 9. The small adjustments are to be expected since the peripheral unit's activation is almost zero and hence Δw will also be near zero. One possible method to expedite this process would be to enforce a minimum weight update and thus allow distant units to make small but more substantial adjustments during each epoch. Such a scheme has yet to be evaluated in practice.

Data distributions with well separated Gaussians are very easy to solve since it is obvious, both to the network and a human observer, that there are two distinct clusters in the data. This is not the case for somewhat more complicated distributions of overlapping Gaussians. Obviously, the higher the degree of overlap between two clusters the more difficult it will be to distinguish them from each other. To analyse this situation additional simulations were conducted on several data distributions containing four Gaussians with varying degrees of overlap between two of the four clusters. Figures 9–11 show these distributions. It was found that both SCL and HCL were able to identify the four clusters but that the convergence times increased with the degree of overlap. Training times on these three datasets and the previous two gaussian dataset are summarized in table 1 below.

[illegible]	[illegible]	[illegible]
[illegible]	4	5
[illegible]	5	7
[illegible]	27	18
[illegible]	32	23

Table 1: Relative training times of HCL and SCL expressed in epochs.

Figures 12 and 13 show the relative convergence times of these algorithms on the initial two Gaussian problem and the four Gaussian problems. As can be seen, SCL was found to converge significantly faster than HCL in the presence of overlap. This

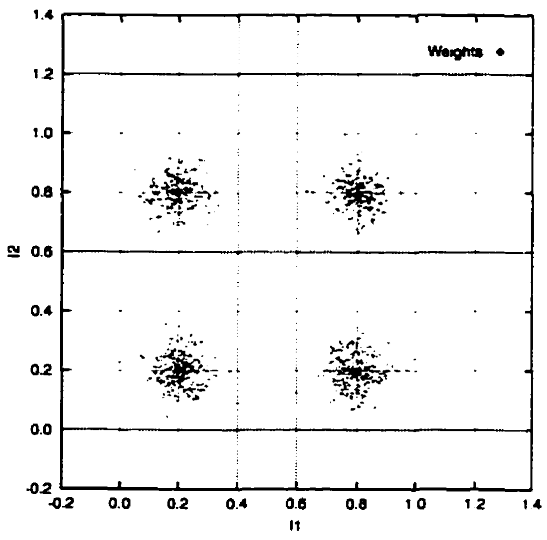


Figure 9: Dataset with four isolated Gaussian clusters and learned cluster centres.

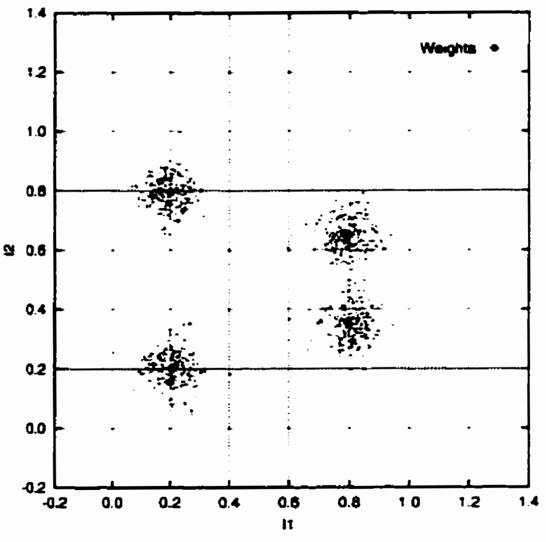


Figure 10: Dataset with four Gaussian clusters displaying slight overlap and showing learned cluster centres.

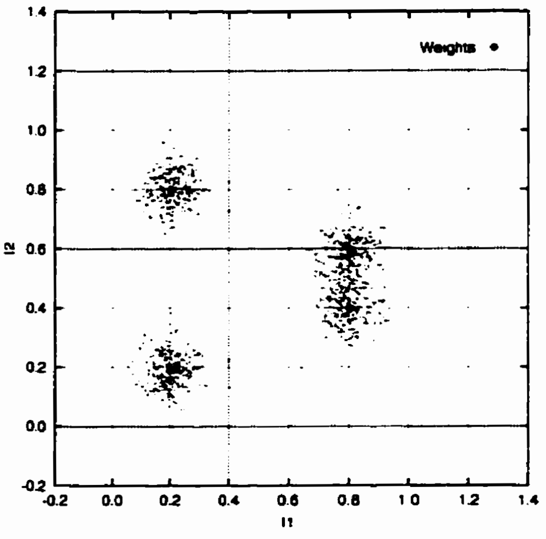


Figure 11: Dataset with four gaussian clusters displaying significant overlap and showing learned cluster centres.

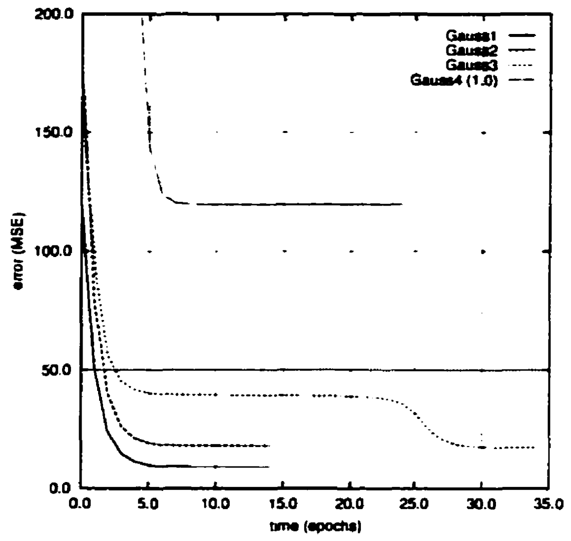


Figure 12: Mean squared error vs time for the HCL algorithm.

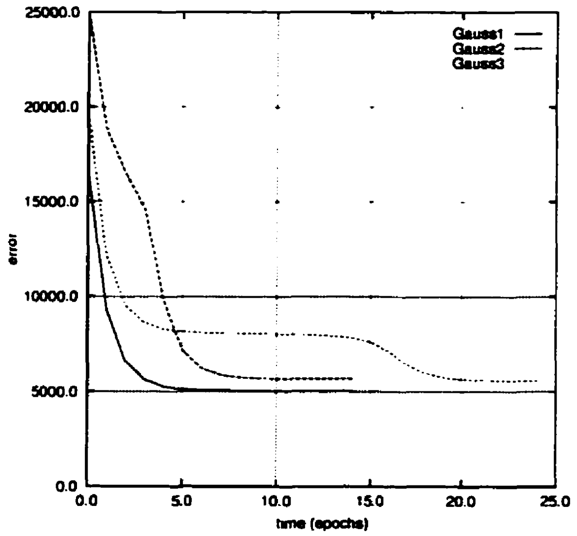


Figure 13: Error vs time for the SCL algorithm.

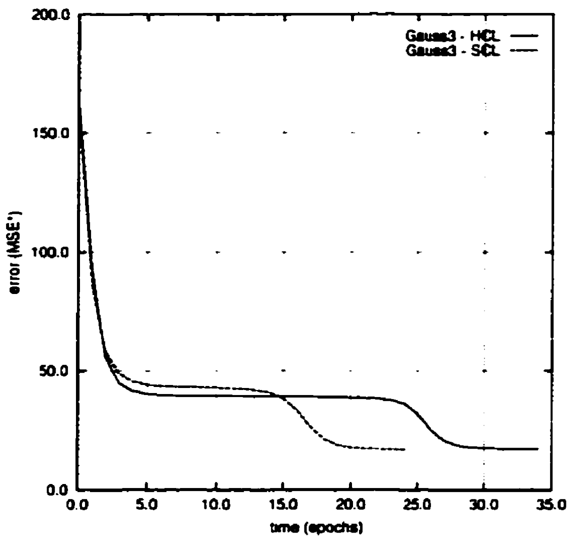


Figure 14: Error versus time comparison for HCL and SCL on the partially overlapping Gaussian problem.

behaviour is more clearly demonstrated in figure 14, which shows the relative training times of HCL and SCL on the "Gauss 3" problem corresponding to figure 10. Here the error values produced by the SCL network are converted into an equivalent mean-squared-error based on the known properties of the data distribution. It should be noted that while SCL provides better performance than HCL, it does so at the expense of additional computation. For the two input, four output network examined here, SCL required approximately five times more computation than HCL.

In most training trials SCL was able to correctly identify the presence of four clusters in spite of the overlap. Yet, in a couple of attempts the network becomes trapped in a non-optimal solution with the two overlapping clusters being covered by only a single unit and the other two clusters being shared by the remaining three units. Though this is not the best possible solution, it can not be considered a complete failure either, since the network was still able to extract some useful information from the data. In contrast, the sensitivity of HCL to its initial weight values makes it quite susceptible to the orphaning of units, which produces solutions that do not represent the data in an acceptable way. For example, the fourth curve in figure 12 shows the

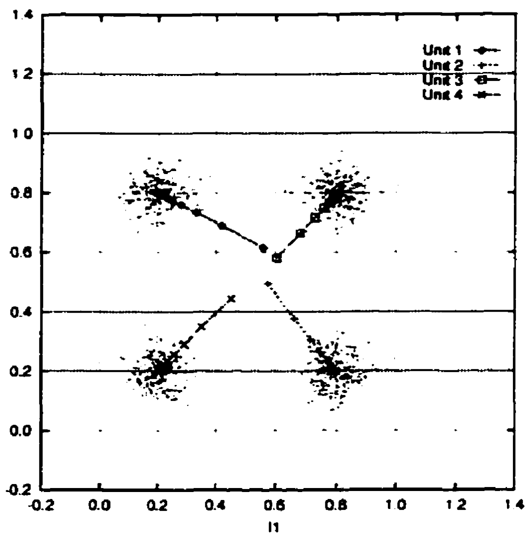


Figure 15: Weight trajectories for an HCL network.

high mean-squared-error resulting from poor initialization of the HCL network's weights. In this case the weights were initialized around 1.0, instead of 0.5 as was done in previous experiments. The result is a mean-squared-error of 119 which clearly indicates the poor quality of the resulting solution. The plots of figures 15 and 16 demonstrate typical weight trajectories through the data space for a successfully trained HCL and SCL network.

3.1.1 Examination of Complex Two-Dimensional Data Distributions

While the previous four examples illustrate the general behaviour and relative performance of the algorithms, one would not consider these discrimination tasks to be particularly challenging. To better gauge the operation of the algorithms on more difficult tasks a more complicated dataset was constructed. This dataset consisted of five Gaussian clusters with varying degrees of overlap and density, and is shown in figure 17. Each of the four clusters to the right of the figure contain 500 elements, while the larger cluster on the left contains 2000. This dataset was presented to a two input, ten output network with initial weight values randomized around 0.5. As

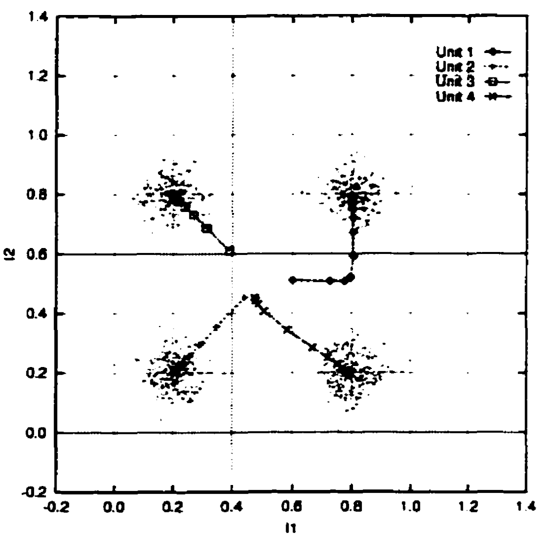


Figure 16: Weight trajectories for an SCL network.

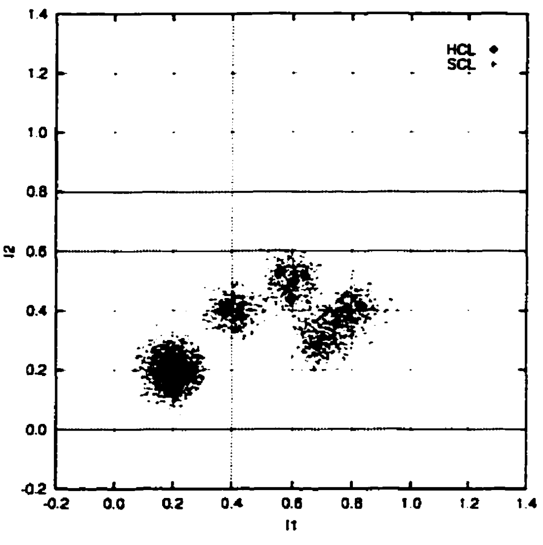


Figure 17: Complex datatset and the resulting HCL and SCL solutions for a 10 unit network.

before, a learning rate of $\varepsilon = 0.001$ was used. The resulting solutions produced by both HCL and SCL are also shown in the same figure.

In examining the solutions one observes that the HCL algorithm distributes its units in such a way as to cover the data in a roughly uniform fashion. Note that the large cluster on the left contains no more units than the smaller cluster immediately adjacent to it, demonstrating that HCL places no significance on the density of data points (as one would expect). However, the solution produced by SCL is quite different. This algorithm places two units at the centre of each of the five clusters. This difference in solutions is a direct result of the differing objective functions being used by the two networks. HCL is attempting to locate the best solution by minimizing the mean-squared-error between its weight vectors and the data vectors, which it can accomplish by spreading the units throughout the data, as shown. SCL, in contrast, is attempting to maximize the probability that its radial-basis-function units are responsible for producing the data values. Placing its units in the manner used by HCL would not accomplish this. Instead the optimal solution under this criterion is to locate the unit centres exactly in the centre of the individual data clusters. This is true even if the result is the coincident placement of units, as was found in this example. Again, it should be noted that this algorithm also places no significance on the density of the data clusters.

In addition to the HCL and SCL algorithms, this dataset was also tested using the Krishnamurthy and DeSieno versions of FSCL. The resulting solutions achieved by these two algorithms are presented in figure 18. It can clearly be seen that both techniques are distributing their units according to the density of the data. The placement of units for the four low density clusters is relatively equivalent. Where the solutions differ markedly is in the placement of units within the high-density cluster. While both algorithms use four units to represent this cluster, $FSCL_D$ places its units at the centre of the data, while $FSCL_K$ distributes them evenly around the cluster. This difference is a direct consequence of the way in which the two networks use the frequency component in determining a solution. For $FSCL_D$ the frequency based bias term (equation 5) strongly influences the selection of a winning unit when

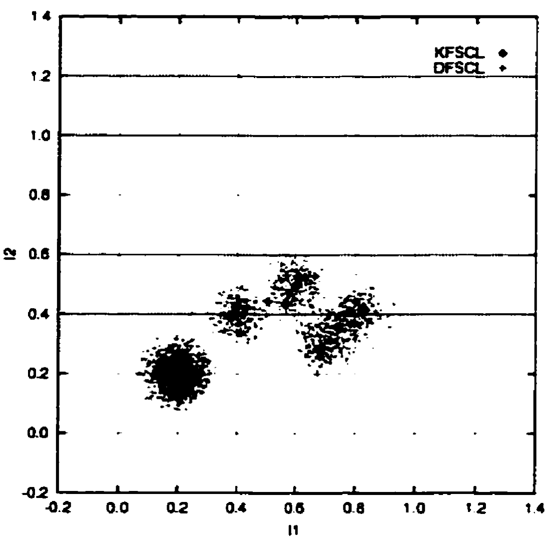


Figure 18: Complex dataset and the resulting $FSCL_K$ and $FSCL_D$ solutions for a 10 units network.

the units are winning a disproportionate number of times. However, once weights have been adjusted in such a way as to result in uniform winning proportions for all units, the bias term no longer dominates the learning and the simple Euclidean distance is used. Under this condition the units can minimize this distance only by moving to the middle of the data cluster.

In contrast, $FSCL_K$ maintains the frequency dependent aspect of the training throughout the entire learning process. As a result, the four units in the dense cluster spread out uniformly in order to assume responsibility for an equal proportion of these data points. This is clearly the superior solution since it makes the best use of all units. To confirm this conclusion, the relative mean-squared-error of HCL, $FSCL_D$, and $FSCL_K$ was recorded during the training process, and the resulting plot is shown in figure 19. The two $FSCL_D$ curves represent the solutions achieved by this algorithm using a bias factor of $C = 1.0$ and $C = 10.0$ respectively. As is quite evident from those two results, the performance of $FSCL_D$ is strongly dependent on the choice of bias factor.

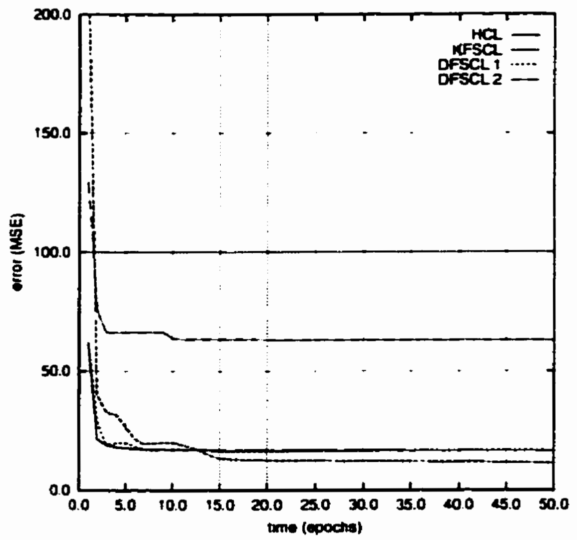


Figure 19: Relative MSE performance of HCL, $FSCL_D$ and $FSCL_K$ on the complex clustering task.

Since $FSCL_K$ produced such an effective solution when confronted with this complex data distribution, we decided to test its ability to solve a second, more challenging problem. This new dataset, shown in figure 20, contains three Gaussians of varying size and density, along with an overlapping rectangular region of uniformly distributed random points. The large Gaussian contains 3000 points, while the medium and small Gaussians contain 1500 and 500 points respectively. The rectangular regions in the lower right is made up of 2000 points, for a total of 7000 data points in the training set. Weights were again randomized around 0.5 at the start of training and a learning rate of $\varepsilon = 0.001$ was used. A total of twenty units were provided to the network and the final placement of the weight vectors is shown in the figure.

As can be seen, $FSCL_K$ once again produced a well structured solution by uniformly positioning units based on the density of points. In spite of the complicated structure the algorithm converges to a solution in less than 5 epochs. This convergence time was largely unaffected by changing the random initial values of the weights. This is

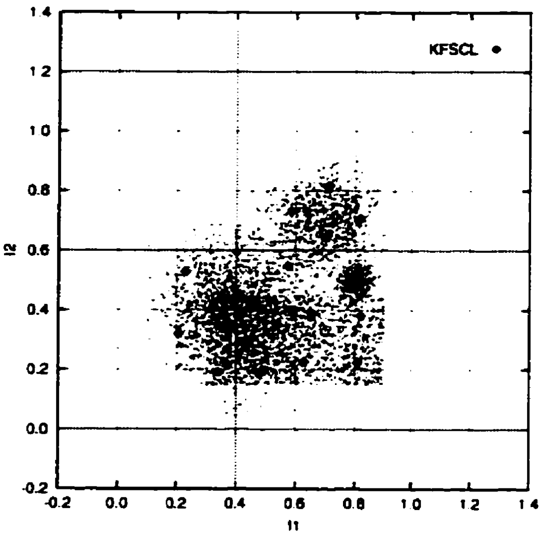


Figure 20: $FSCL_K$ solution to a complex data distribution containing both Gaussian and uniform random data distributions.

demonstrated by error measurements of figure 21, resulting from initialization of the network weights around 0.2, 0.4, and 0.6.

It should be noted that the solution produced by this network is not capable of individually identifying the four original distributions from which the data was constructed, since it has no additional information at its disposal on which to base such a discrimination. However, that discrimination would be possible by employing additional *supervised learning*, or by providing the algorithm with supplementary parameters allowing the separation of these base distributions in a higher dimension.

3.1.2 Effect of Learning Rate on the Performance of Krishnamurthy FSCL

To this point a uniform learning rate of $\varepsilon = 0.001$ has been used for all algorithms and simulations in order to permit fair comparison of the networks. The CL algorithms other than $FSCL_K$ will not generally tolerate large learning rates. However, we wished to test the $FSCL_K$ technique in order to determine how susceptible its performance was to the use of these higher learning rates. To this end, the same network was again

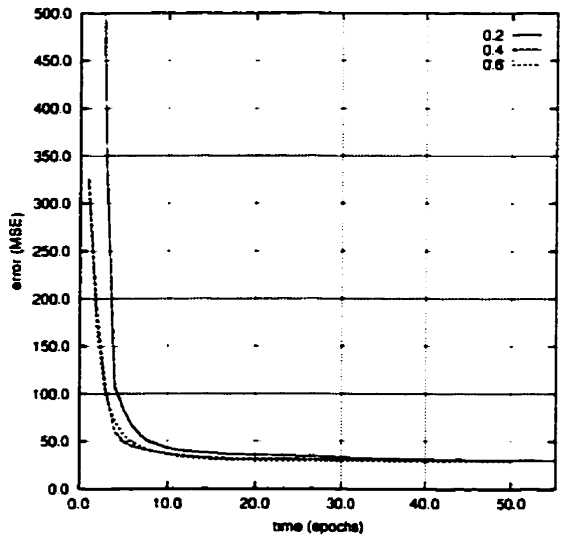


Figure 21: Error performance versus epoch for a $FSCL_K$ network trained on a complex data distribution.

simulated on the second complex data distribution with weights randomized at 0.5, but the learning rate was changed from $\varepsilon=0.001$ to 0.01, 0.1 and 1.0. The resulting mean-squared-error performance versus time is presented in figure 22. As this figure demonstrates, the $FSCL_K$ algorithm is capable of easily locating a stable solution for learning rates up to $\varepsilon=0.1$. However, in the case of $\varepsilon=1.0$ the resulting weight adjustments become so large that the network is unable to converge to a single stable solution. It instead oscillates between many sub-optimal solutions. In spite of this, it is evident that the already expedient learning observed with this algorithm can be safely improved by using moderately larger learning rates than those employed in earlier experiments.

3.2 Effects of Exponential Approximations on Learning Performance

The simulations performed earlier in this chapter show that SCL has a very definite advantages over HCL, both in speed of convergence and quality of solution. However, the arithmetic computations for SCL are more complicated, requiring the evaluation of an exponential function in determining a unit's activation. This

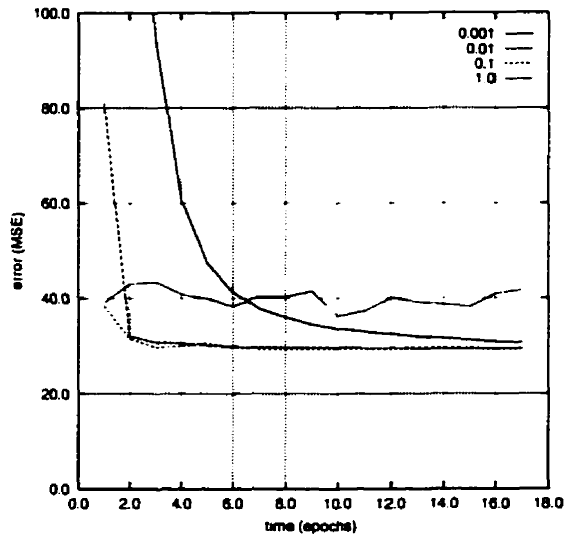


Figure 22: MSE versus time for a $FSCL_K$ network learning a complex dataset using various learning rates.

function is not available in the restricted mathematical repertoire found in most embedded microcontroller applications. As an alternative in these situations it is possible to replace the exponential with a look-up table approximation. However, it is not clear to what degree the use of a look-up table will impact the ability of the algorithm to reach a suitable solution. In order to answer this question, a number of simulations were conducted using the same set of test cases described above, but replacing the exponential function with look-up tables of various sizes. To help improve the accuracy of the approximation, the function $e^x \cong (1 + x)$ was used for arguments in the range [0,−1) and a simple look-up table used for values in the range [−1,−40). All arguments below −40 were considered to be equal to −40. The results of these simulations are summarized in table 2.

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	5	5	5	5	5
[illegible]	3	5	5	5	5
[illegible]	5	2	4	3	0
[illegible]	5	2	4	3	0
[illegible]	4	5	3	0	0

Table 2: Number of correct solutions learned for different sized look-up tables.

Each of the test cases was simulated a total of five times with each size of look-up table. For reference purposes the network's performance using a true exponential is also included in the table. The table values show the number of "correct" solutions discovered by the network in those five learning trials. A solution is considered correct if the network was able to place a prototype at the centre of each of the input clusters. From the table it is clear that larger table sizes provide better performance. However, it should be noted that the network will occasionally get stuck in a sub-optimal solution, independent of whether a true exponential or a look-up table is used.

One other significant side effect of using a look-up table approximation is the network's inability to converge to a stable solution. In those situations the coarse nature of the look-up table results in the network oscillating around the precise solution. This occurred consistently for the 15 and 8 element look-up tables, but was almost nonexistent for the smaller tables. As well, the effect was only evident on the more difficult problems consisting of overlapping Gaussians. The reason for this behaviour is quite clear. The learning algorithm is attempting to make small refinements in the weights in order to move them closer to the centre of the clusters and thereby reduce the network error. However, since there are only a limited number of adjustment values available from the table, the network is unable to make precisely the update it requires and overshoots the desired value. On the next epoch it must then correct for this new error which again results in an overshoot. So the weights end up oscillating back and forth around the true minimum. This effect occurs mainly in the more difficult training situations because these are the cases which require the finest weight adjustments to ensure proper discrimination of the clusters. Even though these oscillations do occur, they are relatively small and do not have a significant effect on the quality of the solution. In tests with the very small look-up tables there are so few values available that the network defaults to making the smallest possible update and eventually reaches a stable solution.

3.3 Determining the Required Number of Network Units

As was discussed in chapter 1, unsupervised methods are ideally suited to situations where the number of data clusters is not known prior to training. The question then arises: How does one know how many output units, and hence receptive fields, to provide in the network for a given problem? There is no easy answer to this question. Up to this point most simulations have been using exactly the same number of units as clusters in the data. An exception is figure 20, where an excess of units were properly employed to represent the data following learning. But providing the optimal number of units is only possible if we know in advance how many clusters exist and in many situations this information is simply not available. What is clear is that providing the network with too few units will make it impossible to find the best solution. However, it is not obvious what effect providing a surplus of units will have in the general case.

In order to answer this question a few simple SCL simulations were conducted. These involved the clustering of the four isolated Gaussians (figure 9) using either one too many or one too few units. As expected, when the network is operating with one less unit than is ideally required, the network will reach a solution in which one unit takes a position between two clusters and attempts to represent both of them. The other units position themselves as before, in the centre of each of the remaining two clusters. Figure 23 shows one such possible solution. For situations where an additional unit is provided the network represents three of the clusters using three of the units, and the two remaining units share responsibility for the fourth.

This behaviour appears to show that the network is capable of dealing in a reasonable way with any extra or even deficient resources at its disposal. It suggests that if the

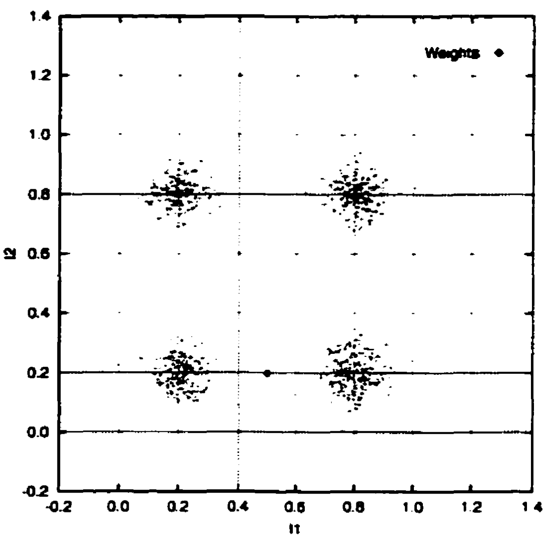


Figure 23: Clustering of four gaussians using three units.

desired number of clusters is not known in advance it is best to provide the network with a larger number of units and perhaps remove redundant units near the end of training through pruning. This would ensure that the network has the necessary resources to discover the true structure in the data, yet only retained the necessary units once training was concluded. Of course, it should be noted that adding extra units will not only retard the learning process, but may also result in the network overfitting the data. As a consequence, the network will achieve better performance on the training data, but poorer generalization. In any case, there appears to be some flexibility in the number of units one provides.

Hardware Systems Employing ANNs 4

Having examined the basic performance of competitive learning in the context of low dimensional input spaces, we would now like to extend these investigations to more practical problems in higher dimensions. Many of the tasks described in the introduction of chapter 1 would be suitable candidates, but of particular interest to this author is the potential benefits of using neural technology as a component in the control of mobile robotic systems. This application is made even more interesting if the robotic systems are constructed from inexpensive, off-the-shelf components. Under these conditions the neural algorithms are subject to additional computational and energy consumption constraints which do not come into play when one uses the algorithms on high-end workstations or in advanced desktop computing environments.

Is it possible to construct systems which are capable of performing meaningful adaptive signal processing tasks using inexpensive, off-the-shelf components? Are the competitive learning algorithms capable of operating under such conditions? What are the consequences of limited precision computations and limited memory resources? It is these questions which we will attempt to answer in the remainder of this thesis.

4.1 Custom Neural Circuitry

A significant portion of the research conducted in our laboratory in recent years has focused on the implementation of neural network algorithms in compact low-power custom VLSI hardware.[15-25] The bulk of this work has concentrated on custom analog neural circuits, though pulse stream digital networks have also been investigated in our laboratory.[20,21] The main advantage to all these approaches is the high synaptic density which can be achieved in comparison to traditional digital circuit implementations. As well, since each of the synapses is essentially a special purpose processor, operating concurrently with all other synapses, there is a great efficiency of computation. This allows for large effective computation rates using fairly modest structures. Our analog neural circuitry has relied heavily on a CMOS version of the wide-range Gilbert multiplier.[26] This analog multiplier is very compact in comparison to an equivalent multiplier constructed using digital components, requiring as few as 19 transistors.

As a result of the long term work, a significant amount of experience has been acquired with respect to the properties of the Gilbert multiplier and how it performs both theoretically and in an actual circuit environment. Past studies have included a detailed investigation into the various types of circuit and fabrication difficulties that would be encountered as a result of implementing an analog system using these multipliers. It was found through those investigations that the analog neural circuits are quite robust and are capable of learning non-trivial tasks while enduring significant fabrication and environmental limitations. Readers interested in a more detailed description of these past results are referred to [15,23].

While the analog circuits have definite advantages in terms of integration density, the technology is not presently available as commercial components. Analog ANNs are still very much a topic of ongoing research. Presently, the only way to exploit this technology is through the design of custom integrated circuits, which certainly violates our requirement for inexpensive systems. In addition to the availability and cost issues, analog neural circuits suffer from a problem common to all analog circuitry, and that is the interfacing of the analog neurons to each other and to other hardware components. Fortunately, due to continued advances in CMOS fabrication technology, the complexity and computational performance of traditional digital circuitry has increased dramatically over the past decade. As well, operating voltages of these devices have decreased, resulting in reduced energy consumption.

Furthermore, the costs of these components has dropped substantially, making it possible to purchase a relatively inexpensive RISC or CISC processor which provides computational power comparable to older custom analog systems. While it is possible to exploit these advances in the analog circuits as well, the speed-up already achieved in the digital systems make them very attractive engines for less aggressive neural applications. Though not as area efficient as a full custom analog implementation, parallel arrangements of digital processors can also provide an effective platform for higher performance neural applications. Such systems may even take advantage of reconfigurable hardware such as field programmable gate arrays (FPGAs) to augment the functions of the serial processors.[27]

The target hardware system selected for this study is that of a simple autonomous mobile robot controlled by an inexpensive digital microprocessor. This system performs limited precision integer computations and has very limited memory resources. In order to determine a suitable sensory arrangement for the robot it was felt that one should look to existing biological systems for motivation as to the type, quantity, and arrangement of sensory inputs. This technique has been used commonly in the past for similar robot applications using non-adaptive control, such as the cricket robot.[39] After some consideration, the vision system of the jumping spider was selected as the basis for the construction of our robotic sensory system.

4.2 Arachnid Biology

Spiders are very familiar creatures to all of us given their presence in all parts of the world. However, due no doubt in large part to our familiarity and occasionally even fear of these creatures, the complex structure and behaviour of these fascinating animals is generally overlooked. One does not typically stop to admire the intricate construction of a spider's web, nor do most people realize that not all spiders build webs. Some species actually hunt their prey instead of trapping it. The well known tarantula is one such example. The ability of a "simple" invertebrate to perform such intricate behaviours gives us some insight into the true complexity of these creatures.

When one does stop to investigate further, one finds that spiders are, in essence, signal processing machines. Their relatively simple collection of neural cells, the ganglia, are responsible for the processing and coordination of a whole host of senses. These senses include tactile receptors in the form of innervated hairs which cover the majority of the spider's body. Most of the hairs provide feedback relating to physical contact with objects, while a few others are so sensitive that they are capable of detecting the motion of minute air currents. This interesting ability permits the spider to sense the motion of prey in its immediate vicinity without making physical contact with it.

In addition to tactile stimuli, there are also chemical receptors located on the first of the spider's four pairs of legs. These receptors provide the animal with a sense of smell, and perhaps taste as well.

Figure 24: Jumping spider indigenous to Manitoba.

Being an invertebrate, the load-bearing structure of the spider is provided by a rigid exoskeleton. In order to ensure that physical stresses do not result in damage to the exoskeleton, the spider has evolved a series of stress sensors which are distributed across the surface of its body. A large proportion of these sensors are concentrated near the joints of the eight legs, since these are regions of high mechanical stress. Not only must the legs carry the weight of the spider's body while it moves across a horizontal surface, but they must also support the animal in a variety of orientations, such as when suspended from the underside of a leaf or hanging from a web. Such conditions may result in considerable stress being applied to the legs, so it is of critical importance to have a mechanism to measure this stress in order to prevent the overloading of these vital members. Also present within the leg joints are a number of proprioreceptors which provide feedback concerning the position of each of the leg joints.

While all but a few species of spiders possess eyes, most species have rather poor vision. Web spiders, for example, receive most of the sensory information they require through the vibration of their webs. Their eyes are necessary only to detect motion, which plays a part in courtship and in reacting to possible danger from predators. Web builders rely mainly on their tactile abilities in determining the location of prey which become entangled in their web traps. As was previously mentioned, there are several species of spiders which do not build webs for trapping prey but instead actively hunt. As one can well imagine, good vision would be a significant asset to those species. As a result, the vision in these spiders is considerably better than their web building cousins.

The group of spiders possessing the most acute eyesight are the jumping spiders (salticidae). While there are many species of jumping spider they all share one common prominent attribute; a large pair of forward facing eyes. Known as the anterior medial (AM) eyes, these eyes provide the spider with superb vision over short distances, which is very important for the identification and tracking of prey. The photomicrograph of figure 25, from our laboratory, shows the frontal view of a typical jumping spider (figure 24) indigenous to Manitoba. One can clearly see the large AM eyes.

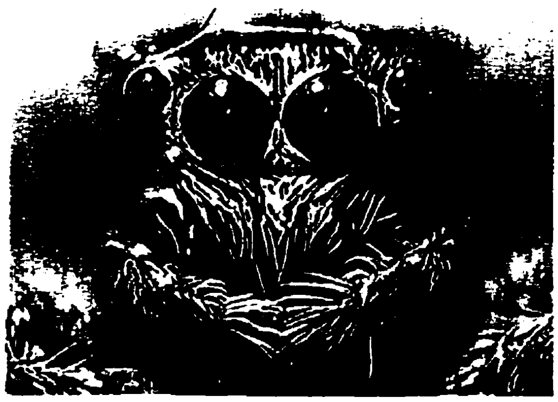


Figure 25: Frontal eye arrangement of a local jumping spider showing the large AM and smaller AL

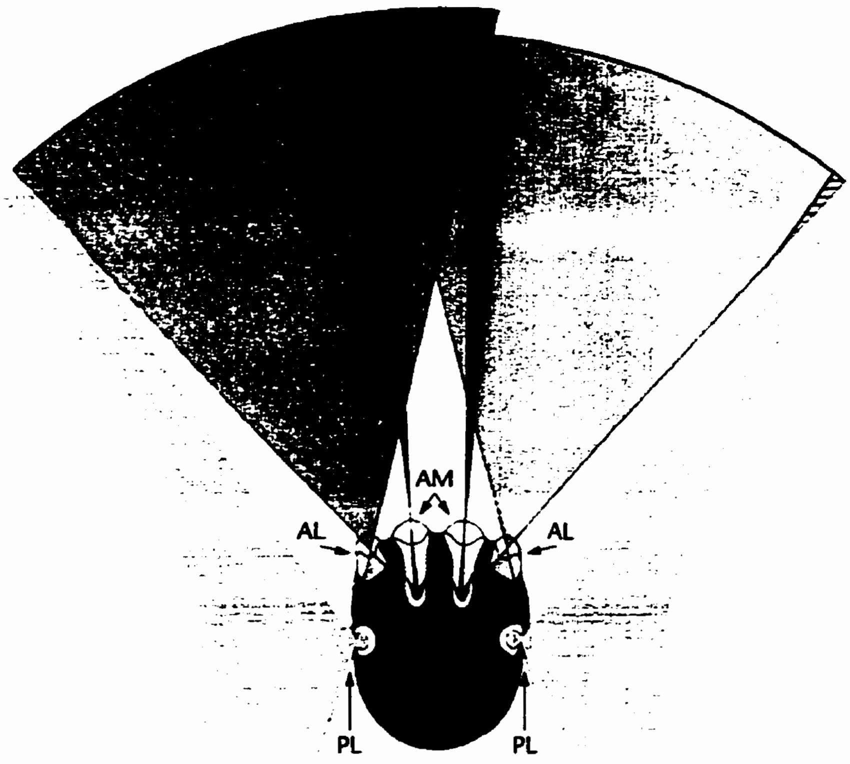


Figure 26: Visual field of the jumping spider.[36]

In addition to the main eyes, the jumping spiders also poses two additional pairs of eyes which are used primarily to detect motion. One of these pairs, the anterior lateral (AL), is located next to the AM eyes on the forward looking surface of the spider's body. These can also be seen clearly in figure 25. The remaining pair, the posterior lateral (PL) eyes, are located on the sides of the body just behind the AL eyes.

Each of the three pairs of eyes provide quite different fields-of-view. Figure 26 shows the arrangement and relative fields-of-view of the spider's six eyes when seen from above. While the four lateral eyes collectively provide a large field-of-view approaching 360°, they do so at the expense of visual acuity. Each PL eye provides vision over an angle of approximately 130°, while the ALs cover an angle of approximately 60° each. There is significant overlap of the ALs at the front of the animal. In contrast to the PLs and ALs, the AM eyes provide detailed vision in only a narrow field of approximately 10°. In order to make the best use of these higher definition detectors, nature has provided the spider with the ability to point the ALs over a range of approximately 70° through the use of muscles attached to the retina at the rear of the eye. Motion detected by the PLs or ALs causes the spider to turn its body to face the object of interest where it can then be examined in detail by the more capable AM eyes.

It is the jumping spider's vision system which has been selected as the basis of the sensory system developed for the autonomous mobile robot which we have constructed during this thesis.

4.3 Autonomous Mobile Robot

The robotic system used in this investigation is shown in figure 27. As can be readily seen from this figure, plastic LEGO® Technic building bricks are used for the mechanical and structural components of this robot. They allow for considerable flexibility in design and construction, while at the same time ensuring that the final

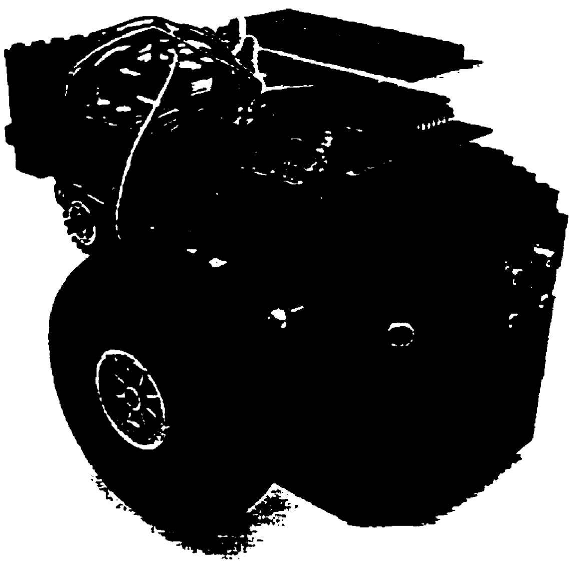


Figure 27: Profile view of second generation LEGO® robot.

system is robust. Control of the robot is provided by a Motorola® MC68HC11 based microcontroller board designed at the Massachusetts Institute of Technology's Media Laboratory.[28,29] This board, named the *HandyBoard*, is capable of receiving input from seven analog and six digital sources, and is able to provide direct control of four DC motors. Conversion of analog inputs to digital values is performed by an A/D converter resident within the HC11 processor. The processor's 8-bit address bus allows for a total addressable system memory space of 64k bytes. However, this address range is shared between support for memory mapped peripheral hardware and 32k of combined program and data RAM. These modest resources significantly constrain the complexity of the neural algorithms which may be implemented on the processor.

While the processor itself is capable of only 8-bit integer computations, a simple multi-tasking operating system and C compiler/interpreter are available for this board, permitting limited operations on 16-bit integers as well as some support for floating point numbers. The compiler/interpreter, known as *Interactive C*, was also originally developed at M.I.T. and is now sold commercially. It allows for convenient programming of the HandyBoard using a subset of standard C programming constructs and conventions.

The robot pictured in figure 27 is actually the second generation of robot designed for use in this work. The original system, shown in Figure 7 of chapter 3, used an earlier implementation of the HandyBoard and provided movement of the robot through the use of a differential tractor drive mechanism, powered by two high-speed DC motors. These motors were connected to the tracks through a gear reduction system which reduced the output speed while increasing the torque provided to the tracks. Unfortunately the gear mechanism, which was also constructed from plastic LEGO gears, suffered from a serious friction problem that ultimately made this design unworkable.

In an effort to improve on the tractor drive it was felt that the simple DC drive motors should be replaced by stepper motors. The operation of a stepper motor is more complicated than a DC motor but provides significantly higher torque at low speeds, thus obviating the need for a gear system. As a result, wheels can be affixed directly to the motor shaft, allowing for a more compact and efficient overall design. In the robot of figure 27 two stepper motors are used, one for each of the two wheels. The use of wheels allows for a straightforward and reliable means of locomotion through the robot's environment. Movement and steering of the robot is achieved through the differential operation of these wheels.

While the DC motors are driven by applying a DC stimulus to a single motor winding, the stepper motors, in contrast, contain four windings which must be excited in a cyclical pattern in order for the motor shaft to maintain a uniform speed of rotation in a particular direction. For the stepper motors used here, excitation of a single winding results in a 7.5° rotation of the shaft. Thus for one complete 360° revolution each winding must be excited in sequence 12 times. If the HandyBoard were required to supply this excitation it would place a significant load on the processor, further restricting the already meagre computational resources available to the neural control algorithm. In order to avoid this impediment, a custom slave controller was designed which provides the necessary excitation to the two stepper motors in response to direction and speed information supplied by the main HC11 processor. This allows the HC11 to issue a single command to the motor controller and then to continue with its regular processing while the slave processor coordinates the low-level operation of the motors. No further intervention is required by the main processor until either the direction or speed of the motors requires adjustment.

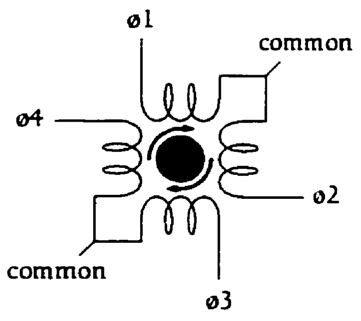


Figure 28: Stepper motor winding arrangement.

The slave controller, pictured in figure 29, receives commands written to an 8-bit register by the main HC11 processor. The controller's PIC16C55 processor then interprets these commands, consisting of a direction bit and 3-bit speed value for each motor. Based on this value an appropriate motor excitation is generated and supplied to the windings through a driver IC. The DIP switches visible at the right of

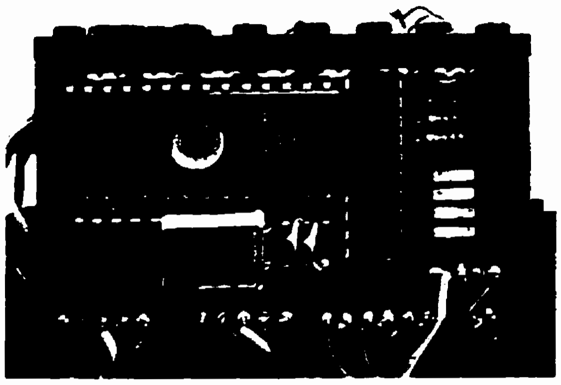


Figure 29: Stepper motor control circuit.

figure 29 permit adjust of the mapping between actual robot speed and the corresponding command byte value.

4.3.1 Robot Sensory System

As was discussed in section 4.2, the biological motivation for the robotic sensory system is the vision system of a jumping spider. As is the case with the spider, the robot's visual experience consists of the combined input from six analog optical detectors mounted on the robot's front surface. Each of these sensors were selected such that their fields-of-view were roughly comparable to that experienced by the spider itself.[1]

The first sensor pair corresponds to the narrow field-of-view of the spider's anterior medial eyes. An Optek OP805 phototransistor was selected here. In order to measure the true angular response of these detectors, a test jig was constructed which permits the detector to be excited by a common optical source and its response recorded. This apparatus, shown in figure 30, allows a source to be placed a fixed radial distance from the detector and moved over an angular distance of ±90° relative to the centre line of the detector. A total of 29 measurements were recorded at fixed intervals over this 180° arc. This process was performed at 7 different radial distances, beginning with a 1cm gap between source and detector, and concluding

1. It is stressed that the complexity of the spider's eyes greatly exceeds that of the optical detectors employed in this robot. However, the essence of the current study is the coordinated interpretation of the detector values.

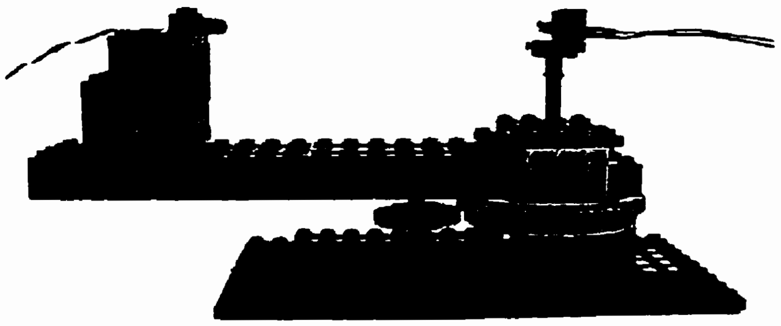


Figure 30: Sensor characterization apparatus.

with a 10.6cm gap. Since the detectors will be connected to the analog input ports of the robot's HandyBoard controller, this board was also used here to perform these measurements. The resulting response of the sensor is shown graphically in figure 31. As can be seen from this figure, the phototransistor provides a strong response over a range of approximately ±15°.

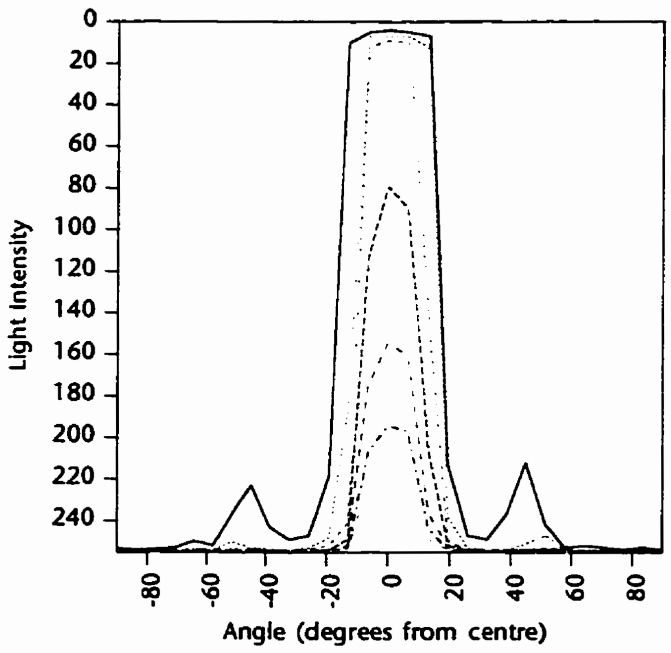


Figure 31: OP805 Phototransistor response vs. angle to source.

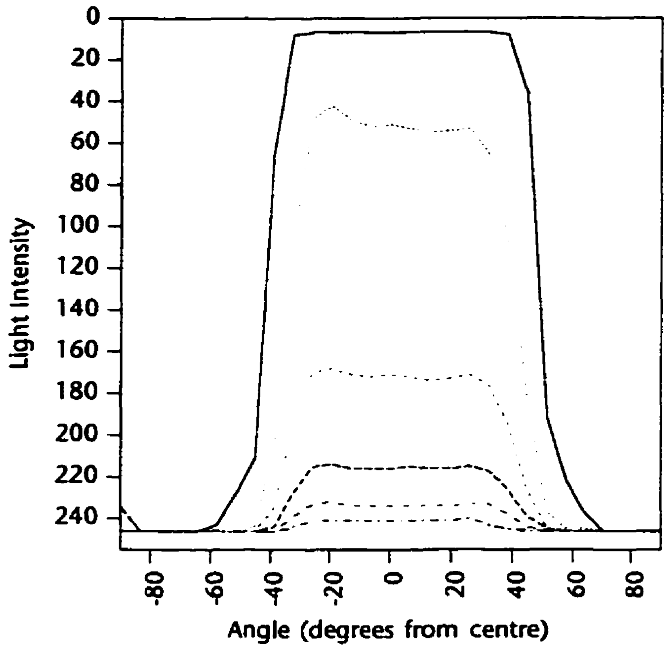


Figure 32: L14C1 Phototransistor response vs. angle to source.

The second pair of optical detectors used were L14C1 phototransistors which correspond to the anterior lateral eyes of the spider. As was done with the OP805 detectors, the response of the L14C1 sensor was tested using the same procedure described above. The response of this sensor is shown in the plot of figure 32. Here the detector shows sensitivity over a range of approximately ±40°.

Finally, a pair of cadmium sulphide photoresistors were used to represent the posterior lateral eyes. Again, the response of these detectors was tested experimentally, resulting in the plot of figure 33. These detectors provide a strong response to optical stimulus over a broad angular range of approximately ±80°. Though not obvious from these measurements, it should be noted that the photoresistors respond much more slowly to sudden changes in light intensity than do the pairs of phototransistors. This behaviour will impede the ability of this

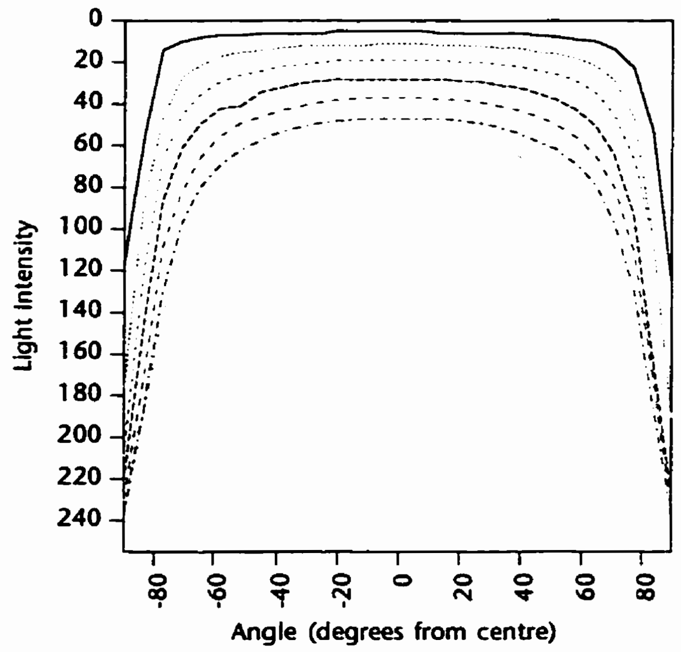


Figure 33: Photoresistor response vs. angle to source.

particular pair of sensors to track rapidly changing sources, should such a situation arise.

The six sensor were arranged on the front of the robot as shown in figure 34.

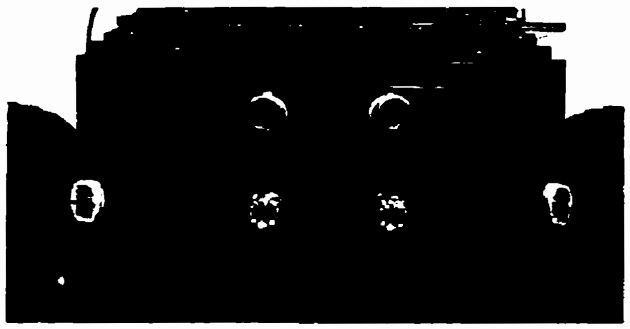


Figure 34: Robot sensor arrangement.

4.3.2 Optical Stimulus Board

Since we employed this robot as a test mechanism for the neural algorithms it was necessary to have some means of providing a well controlled, repeatable optical stimulus to the robot. In order to achieve this, a light panel was constructed which consists of a 5-by-5 grid of light emitting diodes against a contrasting matte black background. The light board is controlled by a custom designed microcontroller board also constructed by the author using a PIC16C74 microprocessor. This allows for any or all of the LEDs to be illuminated at any one time. A series of up to 22 of these light patterns can be downloaded to the board from a Macintosh through a standard RS-232 serial interface. Following download, the PIC controller repeatedly cycles through the patterns at regular intervals. The inter-pattern timing may be adjusted under software control. Red LEDs were used in the design of the light board because their wavelength (635nm) most closely matched the peak reception wavelength of the OP805 and L14C1 phototransistors (870nm).

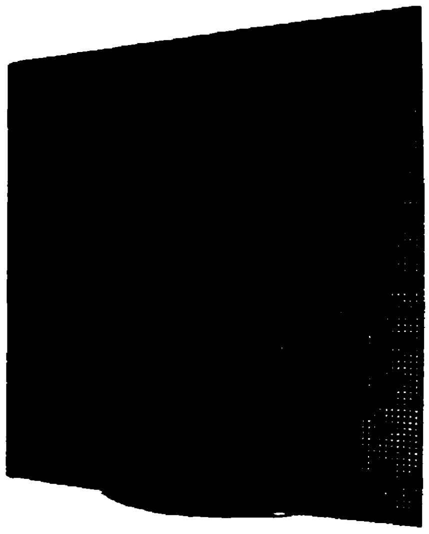


Figure 35: Light board.

Simulations of CL for Robot Vision

5

The algorithmic simulations reported in chapter 3 considered only a two dimensional input environment. Those investigations were then extended into higher dimensional environments based on the visual system of the robot. In place of the simple Gaussian clustering problem used previously, we wanted to select a task which would be appropriate for the robot to perform given the complexity of its optical sensory apparatus. It was decided that detection of both stationary position and directional motion of a single light source would be a useful and challenging behaviour. For the robot to detect the stationary position of a source, all that is required is intelligent processing of the robot's six analog inputs at any instant of time. However, for there to be any possibility of detecting motion, the system will require not only the present readings from its sensors but some time delayed values as well. Based on this requirement the input to the neural network also involved sensor readings with a single time delay, i.e. readings taken at times t and $t-1$, which result in a twelve dimensional input for this particular robot.

Before attempting to simulate learning with the full complex geometry of the robotic vision system, we first investigated the ability of the neural algorithms to cluster both stationary and moving patterns using a somewhat simpler sensor geometry. This geometry was then made progressively more complicated until it mimicked the robot's true sensory apparatus. Once we were confident that the problem was learnable within the constraints of the controlled simulation environment, the same task was tested on the actual robot operating in the real world.

5.1 Identification of Stationary Position

We shall begin by first examining the situation of a sensory system whose geometry closely matches that of the excitation. This would correspond, for example, to an animal which has evolved sensory apparatus that is highly adapted to a specific sensory task. For this investigation the sensor arrangement depicted in figure 36 will be employed. Here the virtual sensors are placed in a cross arrangement with an intersensor spacing of two grid units. All five of these sensors have identical fields-of-view of ±25° from centre and all are oriented with their centres perpendicular to the sensor plane (facing the light sources).

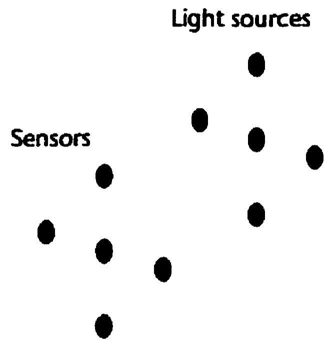


Figure 36: Simple sensor and excitation geometry.

For the purposes of training, an artificial dataset was generated which modelled the output of the five sensors in response to excitation from an array of lights placed a distance of five grid units from the sensor plane. As the figure illustrates, the geometry of the light array in this case matches that of the sensors themselves. Each individual training pattern in the dataset corresponds to a single element of the light array being illuminated. To provide a more realistic model of the sensor response, a small amount of Gaussian random noise is added to the modelled sensory outputs. A training file consisting of 1000 patterns was produced using a standard deviation of 0.001 for this noise. Each of the five lights was illuminated an equal proportion of the time resulting in 200 patterns for each light. The 1000 patterns were then stored in the training file in a random order, thus avoiding the introduction of unwanted systemic effects into the training process.

In addition to the training dataset, a test dataset was also generated which consisted of noise free versions of the five light excitations, as well as four new excitations corresponding to lights placed at the four vacant corners of the light array. This produced a nine pattern test file corresponding to a full 3x3 grid of lights.

A network consisting of five inputs and five outputs was constructed which learned its representation from the training dataset. Initial network weights were randomly selected from a Gaussian distribution with mean 0.5 and standard deviation of 0.1. The learning rate used for all simulations was $\varepsilon=0.001$.

If the network is able to discover the ideal solution, each of the five output units should ultimately learn to represent the excitation from one of the five light sources. That is to say that each output unit should learn to act as a spatial detector for the specific area of the input space in which a light resides. What will be of particular interest is how a successfully trained network responds to the four corner patterns which were not members of its original training set. Will the network be able to generalize on the knowledge gained from the five learned positions in order to provide a useable classification of these four additional patterns?

5.1.1 Hard Competitive Learning

The first algorithm tested on this learning task was hard competitive leaning. As was expected in light of our earlier studies, this algorithm yields consistently sub-optimal solutions. In these solutions a subset of the output units positioned themselves such that they represented the input vectors, while the remaining units were left unused. The precise number of unused units varied depending on the initial values of the weights. With the weights randomized around 0.5 it was typical to only have a single unit or two unused. However, if the weights were initialized around 1.0 or more, only one unit would take responsibility for all 1000 data points, thereby leaving four units unused. This example once again clearly demonstrates that HCL is strongly susceptible to the orphaned unit problem and will generally provide unsatisfactory results when presented with a complex input environment.

5.1.2 DeSieno Frequency Sensitive Competitive Learning

The second algorithm tested was the DeSieno implementation of FSCL. For these simulations each unit's bias distance (B) was set to 0.0001, as before, and a bias factor (C) of 10 was initially used. The result was surprisingly poor network performance.

Repeated training resulted in the algorithm locating the optimal solution in only approximately 50% of the learning trials. Upon further investigation it was discovered that the poor quality of the network's solution was a consequence of improper selection of the bias factor. When this value was changed from 10 to 2 and the simulations repeated, the network was able to consistently locate the optimal solution. However, if the weights are now initialized in an area much farther away from the data, such as around the value 2.0, the network would consistently fail to utilize all units. It was necessary to increase the bias factor needed to near 70 before the network was able to draw all five units in a solution. Unfortunately, the solution achieved under these conditions was completely unusable. Due to the high bias, all units end up selecting identical weight vectors. This strange behaviour appeared to contradict the frequency sensitive nature of the algorithm.

The reason for the unusual learning behaviour is a direct consequence of the way in which the conscience mechanism is introduced into the learning process. The conscience augments the winner selection mechanism of HCL by adding the bias term into the distance computation (equation 7). That is to say, the value which determines which unit undergoes a weight update is the distance from a unit to the input vector, minus the value of the computed bias term. The strength of this bias term depends on the choice of the bias factor. If the bias factor is set too large, it will make an inappropriately large contribution to the distance calculation, dominating the computation for even small differences in the winning proportion of the units. This results in the poor solutions that were initially observed. Conversely, if the bias factor is made too small the conscience will be too weak and will not be capable of influencing the computation enough to avoid the orphaning of units.

This results in a significant problem. The main reason for using the conscience mechanism in the first place is to ensure that all units are contributing to a solution. If the bias factor is set too small this goal is not accomplished. However, if set too large all units are contributing, but the resulting solution is unusable because it doesn't take into account the intricacies of the statistics of the data distribution. This places us in a quandary. The bias factor must be large in order to use all the units, but it must be small to allow those units to learn something useful about the data. A possible compromise may be to initiate the training procedure with a large bias factor to ensure that no units are orphaned, and then to reduce this value as training continues, thus permitting the network to fine tune the solution to better fit the data. However, this would further complicate the algorithm by requiring the addition of yet another adjustable network parameter, the bias factor decay. As well, it is unclear what should govern the initial size of the bias factor when beginning training.

5.1.3 Krishnamurthy Frequency Sensitive Competitive Learning

As an alternative to the DeSieno version of FSCL, the Krishnamurthy implementation was examined. The fairness function used for these tests was $F(u_i) = u_i$, where u_i is the number of times unit i has won a competition. This algorithm was found to work quite well, consistently locating the optimal solution to the task at hand. What makes this algorithm significantly different from the DeSieno technique is the fact that the fairness function is a multiplicative term in the distance calculation, as opposed to an additive one. This allows $FSCL_K$ to avoid the bias factor magnitude problems prevalent in $FSCL_D$.

In situations where the weights are initialized such that they place the units far from the data, $FSCL_K$ will simply require longer training times in order to allow all these units to become included in the solution, but they will definitely all be used. The only major concern here is training long enough for that to take place. A second and perhaps even more important advantage of this technique is the fact that it has no learning parameters which need to be adjusted (except for the ubiquitous learning rate). This significantly improves the reliability of the training process by eliminating additional free parameters from the algorithm.

While $FSCL_K$ successfully learns to identify the five clusters present within the training data, testing with the additional corner points identifies a further difficulty. This lies not with the network's ability to identify the data clusters, but with the winner-take-all form of the output activations. Due to the nature of these outputs, the network must select a single unit as the winner to represent the classification of any input vector. If the vector happens to be only slightly closer to one unit then another, the first unit will assume full responsibility for that vector. As well, if the vector is far from all the clusters but marginally closer to one unit, that one unit will again take full responsibility for the vector and produce an activation of 1.0. This is not the most desirable result because it provides no information as to the confidence the network has in its classifications. To achieve that it would be necessary to replace the winner-take-all nature of the output activations with some form of soft activation. This was done by using the following activation equation:

$$O_i = \frac{\left(\sum_j (x_j - w_{ij})^2\right)^{-1}}{\sum_k \left(\sum_j (x_j - w_{kj})^2\right)^{-1}} \qquad (13)$$

This equation was only used for the computation of the activation value and did not affect the method used in updating the weights. The traditional winner was still used for the purpose of performing these weight adjustments.

As a consequence of this change the network was now able to provide a much more informative and useful classification of the input vectors. This benefit can clearly be seen in figure 37 which shows the network activations using the local representations of the WTA outputs on the left, and the distributed representation resulting from the use of equation 13 on the right. The bottom cells (identical in both diagrams) show the input vector being applied, which is the top-left (TL) source in this case. A full shaded cell in this figure corresponds to an activation of 1.0. The network activations produced by each of the five output units in response to excitation by all nine vectors of the test dataset are given in table 3. Of particular interest is the classification of the four corner patterns which are now represented as a combination of two of the five primitive states. For example, the top-left corner light is being partially

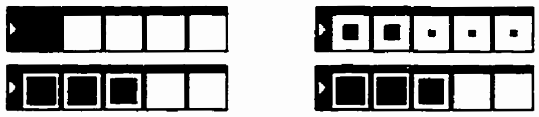


Figure 37: Activations of the $FSCL_K$ network with a local and distributed representation of the same input vector.

	0.0	1.0	0.0	0.0	0.0
	1.0	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	1.0	0.0
	0.0	0.0	1.0	0.0	0.0
	0.0	0.0	0.0	0.0	1.0
	0.342	0342	0.090	0.136	0.090
	0.090	0.342	0.342	0.136	0.090
	0.342	0.090	0.090	0.136	0.342
	0.090	0.090	0.342	0.136	0.342

Table 3: Response from the Krishnamurthy FSCL network employing analog activations.

represented by both the top and left tuned output units. This use of analog activations significantly improves the effectiveness of the $FSCL_K$ algorithm.

5.1.4 Soft Competitive Learning

The final algorithm tested was soft competitive leaning. Again, network weights were randomized to values around 0.5, and the variances of the radial basis functions were initially set to 0.004 to be consist with the earlier low dimensional tests. It was found that SCL was capable of learning this problem but that success in reaching the optimal solution depended on the proper selection of the variance. The initial variance of 0.004 did not provide very good results, but when this variance was increased to values on the order of 0.1 the network was able to easily provide proper clustering of the training data.

When presented with the nine vectors of the test set, SCL was found to produce excellent classifications of the four corner patterns. The inherent analog nature of the Gaussian activations provides a clear indication of the combined classification based on the five primitive states. Typical activations produced by this network for the nine test patterns are given in table 4.

As this learning task has demonstrated, a critical factor in the use of this algorithm is the selection of an appropriate variance for the Gaussian basis functions. If this parameter can be selected appropriately the algorithm performs well. However, as with the choice of the bias factor in the DeSieno version of FSCL, the selection of the variance is actually a two edged sword. This value must be made small enough to ensure that the units span only a single data cluster, while at the same time being

[illegible]	0.0	0.0	1.0	0.0	0.0
[illegible]	1.0	0.0	0.0	0.0	0.0
[illegible]	0.0	0.0	0.0	1.0	0.0
[illegible]	0.0	0.0	0.0	0.0	1.0
[illegible]	0.0	1.0	0.0	0.0	0.0
[illegible]	0.497	0.0	0.503	0.0	0.0
[illegible]	0.0	0.0	0.499	0.0	0.501
[illegible]	0.500	0.500	0.0	0.0	0.0
[illegible]	0.0	0.498	0.0	0.0	0.502

Table 4: Typical activations from the SCL network for the nine vector test dataset.

large enough to make significant advances toward the data in the case when the weights are initialized far from the data. To address this problem, the basic SCL algorithm was modified by introducing a variance decay parameter which will allow the network to begin training with a large variance, and as training progresses, shrink this variance in order to refine the classifications. This modification has been employed by others in the past[30] and is in many ways reminiscent of the neighbourhood technique employed by Kohonen in the self-organizing feature maps. However, as was already discussed, SCL differs from SOFM in that all units adjust their weights at every stage of the learning process, regardless of their location in the network.

5.2 Identification of Object Motion in a Matched Sensory Environment

Having now tested the algorithms on stationary patterns, the learning task was made more difficult through the introduction of motion. To achieve this, a new artificially generated training set was produced based on the sensor and light geometry of figure 36. Each training pattern now consisted of a ten dimensional input vector made up of the five sensor values corresponding to a single light on in one location followed by the five sensor values for a single light on in an adjacent

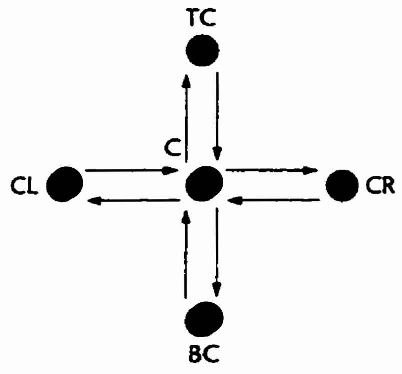


Figure 38: Light transition diagram.

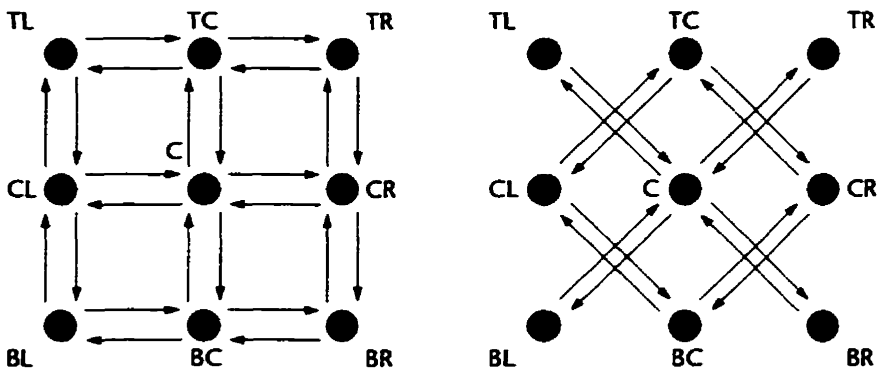


Figure 39: Diagram showing light motion transitions used in testing network generalization.

location. In the case of five lights this corresponds to a total of eight unique light transitions as shown in figure 38. As was done with the stationary light datasets, a small amount of Gaussian noise was added to the calculated values of the sensors in order to more closely mimic the behaviour of a real sensor. A total of 1000 patterns were generated for the training set. These vectors were randomly ordered in the training file so as to remove any undesirable systemic characteristics which may interfere in the training process.

In addition to the training file, three different test datasets were also prepared. The first of these involved the eight noise free versions of the above transitions plus sixteen additional transitions corresponding to the inclusion of the four corner lights. Each of these lights adds two additional vertical and horizontal transitions. The left-hand portion of figure 39 shows all 24 of these (Manhattan) transitions. The second test dataset prepared was intended to test whether the network was capable of not only generalizing to horizontal transitions, but also to diagonal ones. To achieve this, a dataset of sixteen vectors was generated which corresponds to the transitions depicted in the right-hand diagram of figure 39. The final datasets comprised a nine vector file intended to evaluate how a network trained to identify motion interprets the situation where no motion is present. In other words, the ten dimensional input vector consisted of two identical versions of the five sensor values, representing no change in the sensor state from time *t-1* to *t*.

A single layer network consisting of ten inputs and eight outputs was constructed. As before, a learning rate of $\varepsilon=0.001$ was used for all training situations and the network weights were again randomized around the value 0.5 with a standard deviation of 0.1. If the network is able to locate the optimal solution it should tune each of the units to detect one of the eight primitive light transitions.

5.2.1 Hard Competitive Learning

The hard competitive learning algorithm was examined first and found to perform as poorly on the motion clustering problem as it did when confronted with the stationary pattern problem in the previous section. The network consistently failed to make use of all the units in encoding the training data. As before, this problem was exacerbated when the weights were initialized farther away from the data. Based on these test it is clear that hard competitive learning would not be an acceptable learning algorithm for use on this or other similar tasks.

5.2.2 Krishnamurthy Frequency Sensitive Competitive Learning

The second algorithm investigated was the Krishnamurthy version of FSCL. This method was found to very reliably cluster the training data into the eight primitive transitions. As well, the generalization abilities of the network were tested on the twelve new transitions (to and from the four corner points). It was found that the network was able to appropriately represent these new transitions as a combination of the eight learned transitions. The Hinton diagram of figure 40 shows the resulting weight vectors, with a fully shaded cell representing a weight value of 1.0. A list of typical activations resulting from the 24 test patterns is provided in table 5.

As an example, consider the transition from the top-right corner position to the top-centre position (TR–>TC). This transition is represented by a strong excitation of the transitions centre-right to centre (CR–>C), top-centre to centre (TC–>C), and centre to top-centre (C–>TC). With the later two transitions being complementary, one is left to correctly conclude that the primary direction of motion for the light was from right to centre. As well, the fact that the two complementary transitions both involved the top light position allow us to further conclude that the right to centre transition occurred in the top region. It should be noted that though the

0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
0.086	0.175	0.096	0.191	0.191	0.076	0.096	0.086
0.096	0.191	0.191	0.175	0.096	0.086	0.086	0.076
0.086	0.096	0.175	0.191	0.191	0.096	0.076	0.086
0.076	0.191	0.191	0.096	0.175	0.086	0.086	0.096
0.097	0.191	0.086	0.175	0.096	0.086	0.191	0.076
0.191	0.175	0.096	0.191	0.086	0.076	0.096	0.086
0.191	0.096	0.096	0.191	0.086	0.096	0.175	0.086
0.175	0.191	0.086	0.096	0.076	0.086	0.191	0.096
0.086	0.076	0.096	0.086	0.191	0.175	0.096	0.191
0.096	0.086	0.191	0.076	0.096	0.191	0.086	0.175
0.086	0.096	0.175	0.086	0.191	0.096	0.076	0.191
0.076	0.086	0.191	0.096	0.175	0.191	0.086	0.096
0.096	0.086	0.086	0.076	0.096	0.191	0.191	0.175
0.191	0.076	0.096	0.086	0.086	0.176	0.096	0.191
0.191	0.096	0.076	0.086	0.086	0.096	0.175	0.191
0.175	0.086	0.086	0.096	0.076	0.191	0.191	.0.096

Table 5: Activations of a FSCL$_K$ network in response to horizontal and vertical motion patterns.

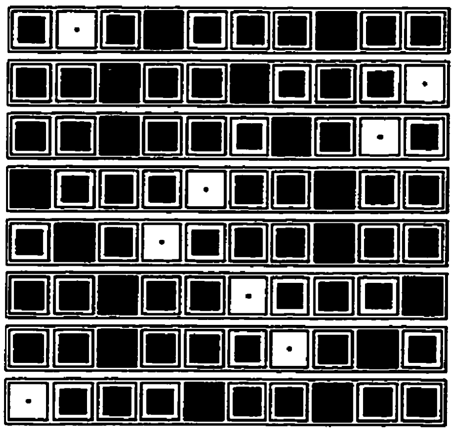


Figure 40: Hinton diagram of FSCL$_K$ network weights following training on the motion detection task.

complementary transitions also both involved the centre position, this does not imply that the transition took place in the centre region because the right to centre transition is a primitive transition and would have been the only active unit had that situation actually occurred.

Based on these very encouraging results the second test dataset containing the diagonal transitions was presented to the network. As with the horizontal and vertical transitions just discussed, it was found that the network was able to generalize very well to these diagonal transitions. When presented with one of the test patterns the network produced a strong activation from two of its units representing the primitive horizontal and vertical transitions which together result in the actual diagonal direction of motion. The complete list of network activations resulting from the application of this training set can be found in table 6.

	0.073	0.073	0.073	0.280	0.280	0.073	0.073	0.074
	0.073	0.280	0.280	0.073	0.073	0.074	0.074	0.073
	0.280	0.073	0.073	0.280	0.074	0.073	0.073	0.074
	0.073	0.280	0.074	0.073	0.073	0.074	0.280	0.073
	0.074	0.073	0.073	0.074	0.280	0.073	0.073	0.280
	0.073	0.074	0.280	0.073	0.073	0.280	0.074	0.073
	0.280	0.073	0.073	0.074	0.074	0.073	0.073	0.280
	0.073	0.074	0.074	0.073	0.073	0.280	0.280	0.073
	0.094	0.094	0.069	0.242	0.094	0.094	0.242	0.069
	0.242	0.242	0.094	0.094	0.069	0.069	0.094	0.094
	0.094	0.094	0.242	0.242	0.094	0.094	0.069	0.069
	0.069	0.242	0.094	0.094	0.242	0.069	0.094	0.094
	0.069	0.069	0.094	0.094	0.242	0.242	0.094	0.094
	0.094	0.094	0.242	0.069	0.094	0.094	0.069	0.242
	0.242	0.069	0.094	0.094	0.069	0.242	0.094	0.094
	0.094	0.094	0.069	0.069	0.094	0.094	0.242	0.242

Table 6: Activations of a $FSCL_K$ network in response to diagonal motion patterns.

Lastly, the $FSCL_K$ network was tested on the final training set which encodes the stationary excitations. In response to these test vectors, the network produced a strong activation for the two complementary transitions representing the TC, CL, CR, and BC positions. For the centre position, the network produced identical outputs from all eight units. In the case of the four corner positions the system produced a strong activation from the four units which constitute the two complementary pairs of transitions corresponding to that particular corner location. The actual activation values themselves are presented in table 7. The fact that the network is able to encode the stationary positions through an aggregation of motion detectors means that a robot (and perhaps even a biological system) does not necessarily require a separate detection mechanism to identify this behaviour.

	0.094	0.242	0.094	0.242	0.094	0.069	0.094	0.069
	0.069	0.094	0.242	0.094	0.242	0.094	0.069	0.094
	0.125	0.125	0.125	0.125	0.125	0.125	0.125	0.125
	0.242	0.094	0.069	0.094	0.069	0.094	0.242	0.094
	0.094	0.069	0.094	0.069	0.094	0.242	0.094	0.242
	0.089	0.161	0.161	0.161	0.161	0.089	0.089	0.089
	0.161	0.161	0.089	0.161	0.089	0.089	0.161	0.089
	0.089	0.089	0.161	0.089	0.161	0.161	0.089	0.161
	0.161	0.089	0.089	0.089	0.089	0.161	0.161	0.161

Table 7: Activations of a $FSCL_K$ network in response to stationary excitation patterns.

Overall, the $FSCL_K$ algorithm performed exceptionally well on all facets of this problem. This outcome is extremely encouraging given the simplicity of the algorithm itself.

5.2.3 DeSieno Frequency Sensitive Competitive Learning

The next algorithm to be tested on the motion identification task was the DeSieno version of FSCL. For these tests the network parameter values $B = 0.0001$ and $C = 2.0$ were used. Weights were once again randomized around 0.5 prior to the start of training. Under these conditions, the network was able to locate the optimal solution on a number of training trials but was quite susceptible to becoming trapped in sub-optimal solutions. As was the case with the tests in section 5.1.2, the quality of the solutions dropped considerably as the weights were initialized farther away from the data. Increasing the value of the bias factor was once again able to draw all units into a solution, but the solution achieved was unusable since all resulting weight vectors were identical.

In those situations where $FSCL_D$ was able to locate the optimal solution its weights vectors were found to be equivalent to those discovered by the $FSCL_K$ network. When the $FSCL_D$ network was modified to use the analog activations of equation 13 the network's performance on the three test datasets was found, in this case, to be equivalent to those produced by the Krishnamurthy technique.

5.2.4 Soft Competitive Learning

Once again, soft competitive learning was the last algorithm to be tested on the current learning task. The initial variance of all units was set to 0.1, with a variance decay factor of 0.995, and a minimum variance limit of 0.0044. Network weights were initially randomized around 0.5. Following training the network was found to have correctly learned to identify the presence of the eight primitive directional transitions, assigning one unit to each. Unit activations in response to the first test dataset, are provided in table 8. Of note is the clearer classifications made by this network in comparison to $FSCL_K$. This is mainly a result of the nature of the Gaussian activation functions used by this technique. In any case, it is easy to see that the solution is functionally equivalent to that achieved by $FSCL_K$.

Testing the network on the diagonal transition dataset also produces roughly equivalent results to those described for $FSCL_K$ with the exception that the activations produced for the two constituent primitive directions of motion are maximally excited, while all other units produce a zero output. This same behaviour was observed when testing on the stationary pattern set. The activations resulting from that test are provided in table 9.

While all networks, except for HCL, were able to locate solutions to this problem, they did not all reach those solutions in the same amount of time. The plot of figure 41 shows the relative mean-squared-error performance versus time for each algorithm. As was done in the low dimensional analysis, the error trace for the SCL algorithm is an adjusted version of the actual SCL error measure, based on the known properties of the solutions space. All four of these algorithms were trained beginning

	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
	0.141	0.0	0.430	0.429	0.0	0.0	0.0	0.0
	0.428	0.431	0.0	0.141	0.0	0.0	0.0	0.0
	0.0	0.141	0.429	0.430	0.0	0.0	0.0	0.0
	0.429	0.430	0.141	0.0	0.0	0.0	0.0	0.0
	0.429	0.0	0.0	0.141	0.0	0.0	0.0	0.431
	0.141	0.0	0.0	0.428	0.0	0.431	0.0	0.0
	0.0	0.0	0.0	0.429	0.0	0.431	0.0	0.141
	0.428	0.0	0.0	0.0	0.0	0.142	0.0	0.430
	0.0	0.0	0.432	0.0	0.427	0.0	0.141	0.0
	0.0	0.429	0.0	0.0	0.140	0.0	0.430	0.0
	0.0	0.141	0.431	0.0	0.428	0.0	0.0	0.0
	0.0	0.430	0.141	0.0	0.0	0.0	0.429	0.0
	0.0	0.0	0.0	0.0	0.141	0.0	0.430	0.430
	0.0	0.0	0.0	0.0	0.428	0.430	0.142	0.0
	0.0	0.0	0.0	0.0	0.429	0.430	0.0	0.141
	0.0	0.0	0.0	0.0	0.0	0.141	0.430	0.428

Table 8: Activations of a SCL network in response to horizontal and vertical motion patterns.

	0.500	0.0	0.0	0.500	0.0	0.0	0.0	0.0
	0.0	0.499	0.501	0.0	0.0	0.0	0.0	0.0
	0.125	0.125	0.125	0.125	0.125	0.125	0.125	0.125
	0.0	0.0	0.0	0.0	0.0	0.501	0.0	0.499
	0.0	0.0	0.0	0.0	0.499	0.0	0.501	0.0
	0.250	0.250	0.250	0.250	0.0	0.0	0.0	0.0
	0.250	0.0	0.0	0.249	0.0	0.252	0.0	0.250
	0.0	0.250	0.251	0.0	0.248	0.0	0.250	0.0
	0.0	0.0	0.0	0.0	0.249	0.251	0.251	0.250

Table 9: Activations of a SCL network in response to stationary excitation patterns.

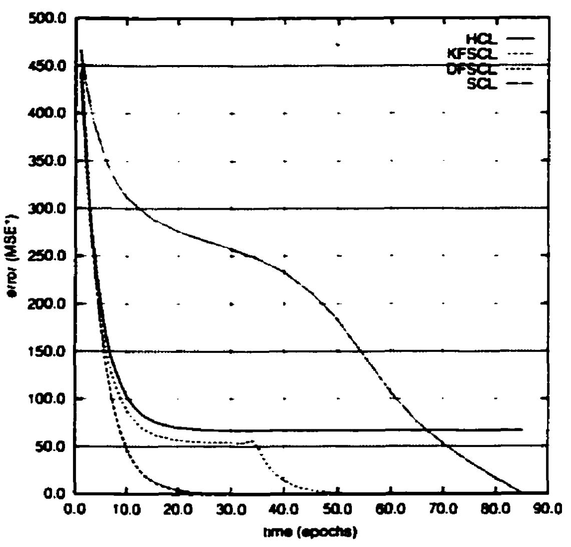


Figure 41: Relative MSE performance of the four learning algorithms with a well matched sensor geometry.*

with the same set of randomly generated initial weights. It is clear from this figure that not only does $FSCL_K$ produce a very good solution (corresponding to near zero error), but it does so very quickly. Convergence is reached in less than 30 epochs, while $FSCL_D$ and SCL required 50 and 85 epochs respectively.

5.3 Identification of Motion in an Unmatched Sensory Environment

Having examined the situation of a sensory system which is well matched to the geometry of the excitation source, attention will now focus on the performance of the algorithms under the condition of an unmatched sensory environment. The sensor arrangement that was used for these experiments is shown in the diagram of figure 42. As can be seen, an additional sensor has been added to bring the sensor count up to six, which is the same number of sensors available on the actual robotic system. However, unlike the robot, the sensors here are still arranged in a perfectly symmetrical manner relative to the excitation. All six sensors possess a ±25° field-of-view relative to the sensor's centre line and all sensors are oriented perpendicular to the sensor plane (facing the excitation).

Once again a training dataset was prepared which included both the current value of the sensors and a single time delayed value. This results in an input vector in twelve dimensions. A dataset of 1000 randomly ordered training patterns was generated, representing the eight transitions between the five basic light positions. Each of these vectors was augmented with a small quantity of Gaussian noise ($\sigma = 0.001$) to provide more realistic variation between modelled sensor values. In addition to the training file, three test datasets were also generated representing the Manhattan transitions, diagonal transitions, and stationary positions respectively for a full 3x3 grid of sources.

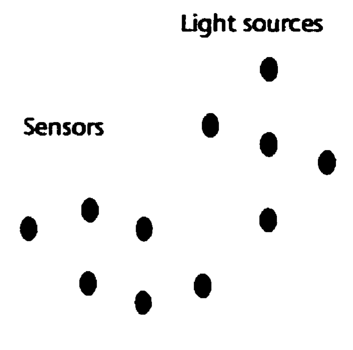


Figure 42: Unmatched sensor and excitation geometry.

5.3.1 Frequency Sensitive Competitive Learning

Given the poor performance of HCL on the more straightforward problems presented in previous sections, we will begin this investigation with the $FSCL_K$ algorithm. The network under test consists of a single layer with twelve inputs and eight outputs. An optimal solution by the network should result in the system once again tuning each unit to act as a motion detector for one of the eight single primitive transitions.

It was found that the algorithm was able to reliably converge to a successful solution in approximately 100 epochs. Following this training the network's generalization performance was evaluated using the three test datasets. As before, the unseen Manhattan transitions were represented by the strong activation of three output units. These corresponded to the primary direction of motion as well as including the two complementary transitions from centre to and from the area of motion. (i.e. For a transition from bottom-left to centre-left, the network excited the bottom-centre to centre, centre-left to centre, and centre to centre-left transitions.) The activation values produced here were only slightly different than those generated by the network when using the well matched sensor geometry.

Tests with the stationary position datasets also produced results consistent with earlier tests. However, in this situation the activations produced for these patterns were not as clearly distinguishable as those produced earlier. Typical activation values are presented in table 10. As can be seen, particularly with the corner positions (TL, TR, BL, BR), the outputs are not as distinct as those presented in table 7.

[illegible]	0.093	0.093	0.093	0.079	0.236	0.236	0.093	0.079
[illegible]	0.214	0.214	0.071	0.107	0.107	0.107	0.071	0.107
[illegible]	0.098	0.098	0.098	0.152	0.152	0.152	0.098	0.152
[illegible]	0.071	0.071	0.214	0.107	0.107	0.107	0.214	0.107
[illegible]	0.093	0.093	0.093	0.236	0.079	0.079	0.093	0.236
[illegible]	0.164	0.164	0.091	0.104	0.141	0.141	0.091	0.104
[illegible]	0.091	0.091	0.164	0.104	0.141	0.141	0.164	0.104
[illegible]	0.164	0.164	0.091	0.141	0.104	0.104	0.091	0.141
[illegible]	0.091	0.091	0.164	0.141	0.104	0.104	0.164	0.141

Table 10: Activations of a $FSCL_K$ network in response to stationary excitation using an unmatched sensor geometry.

In the tests involving diagonal transitions, the network performed very well. These transitions were once again represented by the network as a combination of two primitive transitions; one in the vertical direction and the other in the horizontal. The activations produced for this test were very distinct, more so than for the Manhattan transition tests. In terms of earlier simulations involving the matched sensory system, this network produced stronger activations for some patterns, and slightly less distinct activations for other. Overall the performance of the system on this case is essentially equivalent to the earlier experiments.

5.3.2 DeSieno Frequency Sensitive Competitive Learning

When tested on this clustering task, the DeSieno version of FSCL produced networks with equivalent generalization performance to $FSCL_K$ in the situations where the system was able to successfully cluster the training data. In order to reliably achieve this clustering a bias factor of $C = 10$ was required for weights randomize around 0.5. As was discussed earlier, the reliability of the network in locating a good solution is strongly dependent on the initial values of the weights and selection of bias factor. When the weights were randomized to values significantly larger than 0.5, reliable training was not achievable.

5.3.3 Soft Competitive Learning

Finally, the clustering task was attempted using the SCL network. For this experiment an initial variance of 0.2 and decay rate of 0.998 was used. Both of these values were selected empirically. Under these conditions the network was able to efficiently learn to cluster the training data into the eight primitive transitions. For smaller initial variance, such as 0.1, the system had a tendency to become trapped in sub-optimal solutions.

0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
0.108	0.0	0.001	0.0	0.0	0.108	0.783	0.0
0.108	0.001	0.0	0.783	0.0	0.108	0.0	0.0
0.0	0.0	0.0	0.641	0.0	0.044	0.314	0.0
0.044	0.0	0.0	0.314	0.0	0.0	0.641	0.0
0.108	0.001	0.0	0.0	0.781	0.108	0.0	0.0
0.108	0.0	0.001	0.0	0.0	0.108	0.0	0.783
0.0	0.0	0.0	0.0	0.641	0.044	0.0	0.314
0.044	0.0	0.0	0.0	0.315	0.0	0.0	0.641
0.0	0.108	0.109	0.0	0.0	0.001	0.783	0.0
0.001	0.108	0.108	0.783	0.0	0.0	0.0	0.0
0.0	0.0	0.044	0.641	0.0	0.0	0.315	0.0
0.0	0.043	0.0	0.315	0.0	0.0	0.641	0.0
0.001	0.108	0.108	0.0	0.783	0.0	0.0	0.0
0.0	0.108	0.109	0.0	0.0	0.001	0.0	0.783
0.0	0.0	0.044	0.0	0.641	0.0	0.0	0.315
0.0	0.043	0.0	0.0	0.315	0.0	0.0	0.640

Table 11: Activations of a SCL network in response to horizontal and vertical motion patterns.

Following successful training the network was presented with the three test datasets and found to provide acceptable generalization performance in most cases. However, a few of the activation values generated have the potential of leading to misidentification. The activation values arising from the Manhattan transition tests are listed in table 11. Note that the output of unit 1 for the transitions CL->TL and CR->TR is only 0.044 as compared to the strong output (0.314 and 0.641) produced by the other two units in those rows. Similar behaviour was observed in the classification of the diagonal transitions and stationary light locations, though the smallest output produced in those conditions was 0.063 and 0.061 respectively. Although the network can be considered to correctly generalize to these new vectors, the results must be used with caution in the case of SCL. This is because these small activations can easily be misinterpreted as a lack of excitation.

In summary, the large variety of visual events captured by the raw twelve dimensional sensory data is not an appropriate representation of the environment. The competitive learning algorithms reduce the multitude of possibilities into a limited number of system states, represented by the activations of the competitive units. This is a much better representation of what is occurring in the environment (for example, left to right motion). In the case of local representations, there are only as many states as there are competitive units.

Based on the improved distributed representations, the connection to various behavioural responses (for example, turn right) is learned with supervision in a straightforward manner. The success in learning these representations is much greater for sensors matched to the environment, as one would expect based on observations of animals which have evolved over long periods in a specific environment.

Experiments with the Physical Robot 6

Having successfully shown that the algorithms are capable of correctly learning to classify both moving and stationary patterns in the simplified visual geometries of chapter 5, the investigation next dealt with the geometry of the actual physical robot. We first examined a simulated version of the robot's sensor geometry, then extended that investigation to include operation of the algorithms on the real robot hardware.

6.1 Motion Detection with the Physical Robotic Sensor Geometry

In the previous chapter the simulated sensory geometry was a symmetrical system employing sensors with identical properties. The sensory systems of the robot, shown diagrammatically in figure 43, is symmetric about the vertical centre line but not relative to the horizontal. Each of the grid units within the diagram corresponds to 1cm in the real physical world. The sensor pairs are labelled with the positional names corresponding to the spider eyes which they are representing.

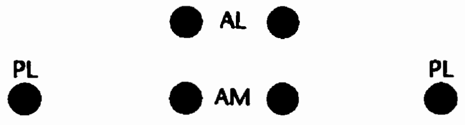


Figure 43: Robot style sensor arrangement.

The anterior medial (AM) pair of detectors is modelled with a ±15° field-of-view relative to the centre line of the sensor, which corresponds to the robot's OP805 phototransistors. Similarly, the anterior lateral (AL) sensor pair sports a ±40° field-of-view and corresponds to the L14C1 phototransistors. All four of these sensors are oriented perpendicular to the sensor plane. The last pair of sensors (PL) are modelled after the cadmium sulphide photoresistors which provide a 180° field-of-view. However, unlike the other sensors this pair is rotated +90° and −90° along its vertical axis and relative to the sensor plane.

This new geometry was first evaluated on the stationary light clustering problem before the task of motion classification was considered. Here the input vector to the neural network consists of the six instantaneous sensor values corresponding to excitation from a cross shaped light arrangement comprising five sources. As with the sensors, the properties of the grid of light sources was modelled after the real physical system which it represents. In the real light-board each of the LED sources is arranged on a regular grid with 3.5cm spacing. The side view of figure 44 illustrates the relative position and view of the AL and AM sensors, and the spacing of the lights in relation to those sensors.

Based on the described sensor-excitation geometry a training dataset was generated consisting of 1000 patterns representing the sensor excitations produced by the five sources. Each pattern vector included a small amount of additive random Gaussian noise. In addition to the training file, a test file was also produced containing the nine sensor excitations corresponding to a full 3x3 grid of lights.

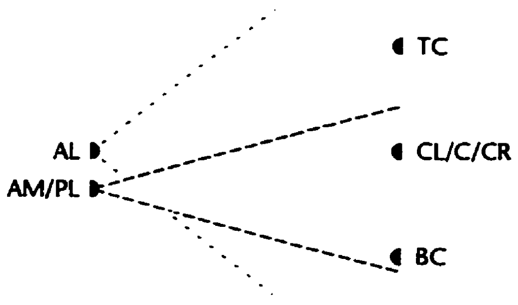


Figure 44: Side view of sensor-light board geometry and fields-of-view.

A six input, five output network was trained using the FSCL$_K$ algorithm. Weights were initialized to random values around 0.5. While the network was able to learn to cluster the training patterns for the five light positions, generalization of the system to the four unseen corner patterns produced potentially ambiguous classifications. The network was only able to provide a clear indication of horizontal position and produced an extremely weak response concerning vertical location. The list of results in table 12 shows the activations obtained for these nine test patterns. Networks successfully trained with the $FSCL_D$ and SCL algorithms gave solutions of equivalent quality.

0.0	0.0	0.0	0.0	1.0
0.0	1.0	0.0	0.0	0.0
0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.0	1.0	0.0
1.0	0.0	0.0	0.0	0.0
0.016	**0.887**	0.015	0.024	0.058
0.016	0.024	0.015	**0.887**	0.058
0.006	**0.964**	0.005	0.008	0.017
0.006	0.008	0.005	**0.964**	0.017

Table 12: Response from the $FSCL_K$ network in the classification of positional patterns using the robot based sensory and light board arrangements.

The system was further tested on the motion detection task. These tests were carried out on a twelve input, eight output network again using the $FSCL_K$ algorithm. The network was found to be capable of clustering the training data but, as with the case with the stationary tests, produced much less distinct classifications of the unseen patterns when compared with the earlier tests of chapter 5. The activations of table 13 show the networks response to those patterns. These values demonstrate that the network is able to extract some general information concerning lateral motion, but is incapable of providing a definitive classification of the vertical component of motion.

The less than stellar generalization performance of the network on the stationary and moving pattern problems is not too surprising when one considers the sensor geometry the system has at its disposal. Due to the narrow field-of-view of the AM detectors and their vertical position relative to the excitation sources (figure 44), they will measure roughly equivalent light intensities for sources in the centre and bottom regions, while producing a zero value for sources in the top region. Furthermore, the left and right sources are also outside of the field-of-view for these

0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
0.108	0.088	0.089	0.109	0.150	0.148	0.198	0.108
0.109	0.089	0.088	0.108	0.148	0.150	0.108	0.198
0.105	0.089	0.089	0.105	0.134	0.133	0.163	0.182
0.105	0.089	0.089	0.105	0.133	0.134	0.182	0.163
0.198	0.089	0.088	0.108	0.148	0.150	0.108	0.109
0.108	0.088	0.089	0.198	0.150	0.148	0.109	0.108
0.182	0.089	0.089	0.163	0.134	0.133	0.105	0.105
0.163	0.089	0.089	0.182	0.133	0.134	0.105	0.105
0.006	0.013	0.013	0.019	0.039	0.008	0.895	0.006
0.019	0.013	0.013	0.007	0.008	0.039	0.007	0.895
0.104	0.089	0.090	0.105	0.132	0.131	0.173	0.175
0.105	0.090	0.089	0.104	0.131	0.132	0.175	0.173
0.895	0.013	0.013	0.006	0.008	0.039	0.006	0.019
0.007	0.013	0.013	0.895	0.039	0.008	0.019	0.007
0.175	0.089	0.090	0.173	0.132	0.131	0.105	0.104
0.173	0.090	0.089	0.175	0.131	0.132	0.104	0.105

Table 13: Activations of $FSCL_K$ network in response to motion patterns.

sensors in this situation. Therefore, the only discrimination that these detectors are able to make is between the top-centre and centre light positions. To exacerbate the network's difficulties in vertical discrimination, the AL detectors receive virtually identical excitations from the top, centre, and bottom sources owing to the distance from the sensors to the source. The final pair of sensors (PLs) are oriented such that they are only able to detect excitations on the left or right. They produce a zero value for excitations in the centre of the light-board. In terms of lateral movement, however, the differential signal from the AL and PL do make it possible to identify motion in this plane.

Based on this analysis it is clear that expecting the network to generalize to the extent that it can detect motion in any region is unrealistic given the current sensor geometry. However, the network does have enough information to allow the system to identify the presence of an excitation and to track its motion laterally. If only the lateral degree of freedom is considered, this is the type of behaviour that a jumping spider will produce.[35]

When testing other algorithms on these training tasks it was found that SCL was also capable of clustering the training data and was able to generalize upon that information in order to detect lateral motion. However, the $FSCL_D$ technique was incapable of learning this same task. The histogram of figure 45 shows the generalization performance of the three networks on the motion detection experiments of the previous chapter when compared to those conducted with the sensor arrangement of the robot. To be considered to have correctly generalized, the pattern of activation values must lie above a single threshold selected across the three test datasets (Manhattan, diagonal, and stationary). This threshold value changes from algorithm to algorithm, but is consistent between the three tests on the same algorithm. The values shown for the DeSieno version of FSCL represent solutions obtained under the best possible weight initialization and algorithmic parameter

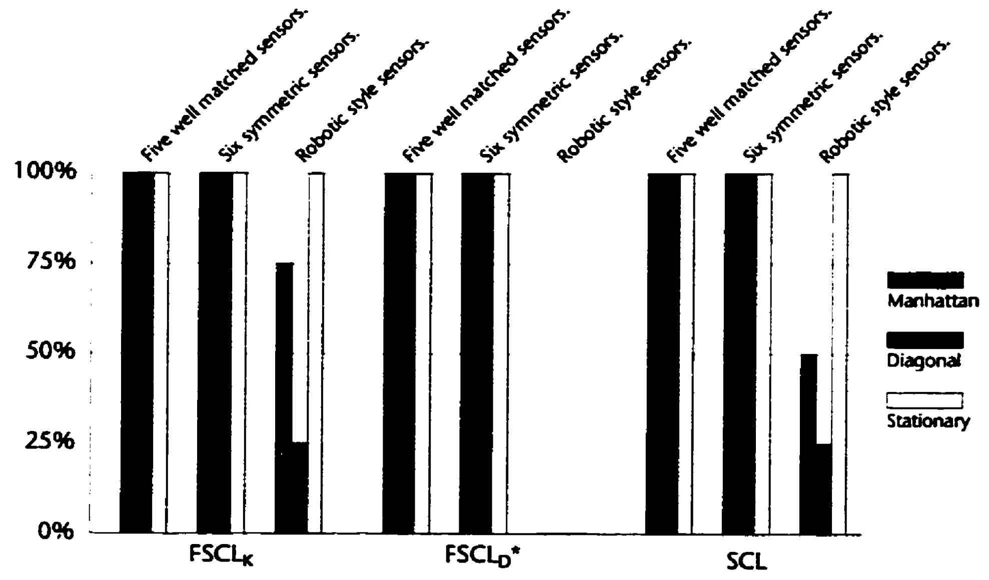


*Figure 45: Generalization properties determined as percent of novel patterns correctly represented. (*Best possible algorithm settings.)*

tuning. As noted previously, without proper adjustment most FSCL_D experiments produce unsatisfactory solutions.

As can be seen from the figure, all algorithms are capable of successfully learning appropriate representations in the first two situations involving the symmetric sensor arrangements. For the third situation employing the robotic style sensors the FSCL_K and SCL algorithms provide closely comparable performance, while FSCL_D fails completely under these conditions.

6.1.1 Learning with a Modified Robotic Sensory System

The inability of the networks to generalize to all aspects of the motion tracking problem is a consequence of the sensor geometry used on the robot. In order to provide the network with the tools with which to extract these properties of its environment, it was necessary to modify the robot's sensor apparatus in order to provide either additional sensors or to change the characteristics of the sensors already being used. It was this later option that was examined.

By changing the lateral orientation of the modelled photoresistors from ±90° to ±45° it was found that the FSCL_K algorithm was able to not only cluster the training data, but also to generalize on this information in order to properly classify virtually all of the test data. Notable exceptions to this were the diagonal transitions involving the bottom source locations. However, for those transitions the network was able to correctly identify the lateral contribution of the transition, only failing to include the vertical component. As well, the activations produced when presented with the unseen transitions were not as definitive as was observed when the sensor geometry was more closely matched to the excitation geometry, though one would certainly expect this to be the case.

The overall mean-squared-error performance of FSCL_K on the motion classification experiments conducted in both this and the previous chapter are summarized in figure 46. As this figure shows, the learning times get progressively longer as the geometry of the sensors deviates from that of the excitations. However, the most

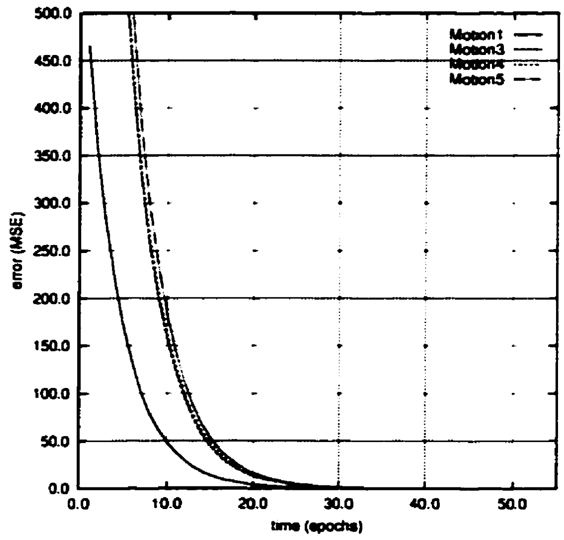


Figure 46: Relative MSE performance versus learning time for a $FSCL_K$ network trained on the various motion detection tasks.

marked change is between the precisely matched and the remaining tests. In all cases the learning times are quite short.

6.2 Learning in the Absence of Floating Point Computations

Throughout this thesis the simulation work presented has involved the use of floating point computations. However, the MC68HC11 processor, which runs the neural learning algorithm on the real robot, does not inherently support floating point operations. As was mentioned in section 4.3, the *Interactive C* programming environment does allow for the emulation of floating point computations on this integer processor, but this significantly retards the learning. The most desirable option is to perform the neural learning using only integer calculations and thereby avoid the overhead of floating point emulation. However, it is not obvious that such an alternative is possible.

In an attempt to answer that question, a specially modified version of the $FSCL_K$ algorithm was incorporated into the neural network simulator environment. This algorithm accepts sensory input in the form of integer values over the range [0,255] and produces "analog" activations in the same range via equation 14.

$$O_i = \frac{255 \times \sum_j (x_j - w_{ij})^2}{\sum_k \left(\sum_j (x_j - w_{kj})^2 \right)} \tag{14}$$

Based on this equation, stronger outputs correspond to smaller activation values. In addition to using integer inputs, the weights stored by the network were also constrained to integer values.

Training and generalization performance of this network was tested on an integer version of the five sensor motion classification problem originally presented in section 5.2. Again, a 1000 pattern training file was generated and presented to a ten input, eight output network. A learning rate of $\varepsilon = 1$ was employed here resulting in very short convergence times (2–3 epochs). It was found that the network was able to successful cluster the training data and thereby identify the existence of the eight basic light transitions. Furthermore, tests of generalization on the Manhattan style transitions involving the corner lights was also performed and these demonstrated that the integer computations did not hinder the network's ability to properly generalize to these previously unseen inputs. The activations provided in table 14 show the actual responses produced by the network. Tests of diagonal transitions and stationary position yielded comparable results.

The above process was repeated using the robotic sensor geometry described in section 6.1.1. Once again, the network was able to successfully cluster the training data and to generalize upon that knowledge in the classification of the Manhattan transitions and stationary position tests. Performance on the diagonal transitions was comparable to that achieved with floating point computations, but again showed difficulties in dealing with transitions involving the bottom row of lights. This is due mainly to the sensor geometry and does not identify a deficiency of the algorithm itself or in its integer based implementation.

19	37	37	29	0	46	29	37
29	0	46	37	37	37	37	29
37	37	37	46	29	29	0	37
0	29	29	37	37	37	37	46
46	29	29	37	37	37	37	0
37	37	37	0	29	29	46	37
29	46	0	37	37	37	37	29
37	37	37	29	46	0	29	37
36	19	45	40	18	40	18	36
18	18	40	36	19	45	36	40
19	36	36	40	18	40	18	45
18	18	40	45	36	36	19	40
40	18	40	36	19	45	36	18
36	19	45	18	18	40	40	36
45	36	36	18	18	40	40	19
40	18	40	19	36	36	45	18
36	45	19	40	40	18	18	36
18	40	18	36	45	20	36	40
19	36	36	40	40	18	18	45
18	40	18	45	36	36	19	40
40	40	18	36	45	20	36	18
36	45	19	18	40	18	40	36
45	36	36	18	40	18	40	19
40	40	18	19	36	36	45	18

Table 14: Activations produced by an $FSCL_K$ network employing integer computations.

6.3 Neural Learning in Robotic Hardware

Having exhaustively tested the learning capabilities of the neural algorithms in simulated environments, we were in a position to incorporate one of these algorithms into the actual physical robot. Based on the results already reported, it is clear that the $FSCL_K$ algorithm is the most reasonable algorithm to implement in the real robotic system. It has consistently proven its ability to converge to an optimal solution, and is straightforward to implement. To this end, a version of the algorithm was coded in *Interactive C* and downloaded to the robot for evaluation.

Training was conducted in a dark room to avoid the ambient light interfering with the excitations produced by the light-board. Figure 47 shows the experimental setup used. The robot was positioned a distance of 10.7cm from the light-board and a

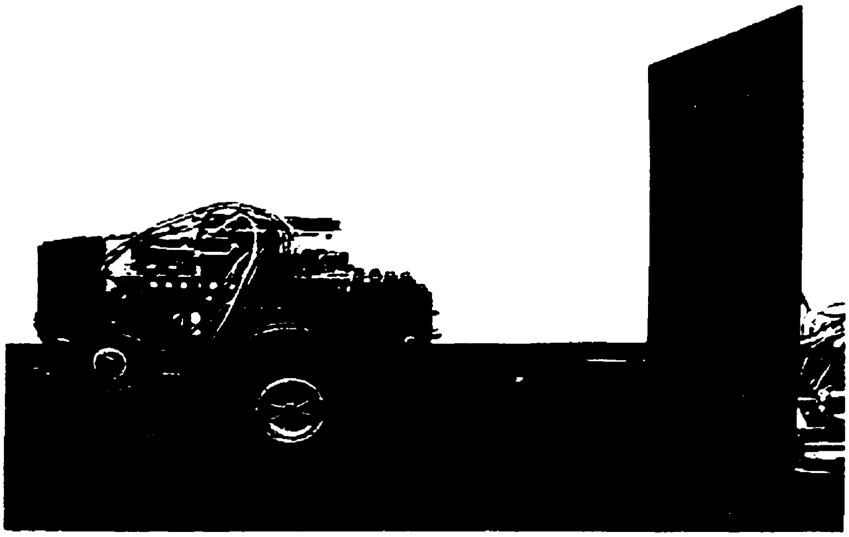


Figure 47: Training of the robotic system.

matte black surface was placed under it and the robot to minimize spurious reflections. The light-board was configured to iterate (sequentially) through the standard eight light transitions. In order to ensure that the network only trained on the transitions and not on stationary excitations, the system performed a simple comparison of the t and $t-1$ sensor values and only passed these values on to the network when they were found to differ from each other. A small threshold value was used here to keep sensor noise from being interpreted as a transition.

The robot was trained in real-time on a total of 2000 iterations of the eight source transitions. Following training, the resulting weight vectors were downloaded to a PowerBook computer via the HandyBoard's serial communications interface for later analysis. In addition to this information, 1000 raw input vectors from the robot were also captured and transmitted to the PowerBook. These vectors were later supplied as inputs to the neural simulator running the integer version of the $FSCL_K$ algorithm. The simulated network was trained for two epochs on the 1000 pattern and the resulting weight vectors were then compared to those received from the robot. This comparison showed that both systems produced almost identical weight vectors. The fact that the vectors do not match exactly is due to the fact that the two networks were not trained on precisely the same 2000 data patterns. As well, during training of the simulated network it was observed that the weight vectors undergo a small amount of cyclical oscillation. This is most likely due to the ordered presentation of data patterns, as opposed to the random ordering use in other simulations.

Following training the light-board was reconfigured to allow for presentation of the 24 Manhattan test transitions. Due to the nature of the robot's operating environment it was difficult to directly observe the robot's response to these test patterns. To facilitate a more straightforward evaluation of the learned solution, the robot's raw sensory response to the 24 test patterns was recorded and communicated to the PowerBook. Since it was found that both the actual robot and the simulated version closely agree, the recorded test patterns were presented to the simulated network and the resulting activations examined. These activations can be found in table 15. As this table shows, the networks were able to clearly learn six of the eight transitions present in the training data, but had difficulty with those transitions involving the top source. The generalized responses show correct lateral classification but rather ambiguous vertical classification. This behaviour is attributed to the poor dynamic range of the AL phototransistors. Without a significant contribution from these sensors it is virtually impossible for the network to extract vertical information from the observed patterns. The AM sensors provide some information owing to their limited field-of-view, but this turns out to be insufficient.

In summary, the computational limitations of the physical mobile robot implemented here were not responsible for the robot's difficulties in learning the above problems. The $FSCL_K$ learning algorithm performed well in spite of both the limited precision and limited memory resources available. Instead, it was the sensory system which ultimately curtailed performance.

35	25	27	28	25	20	41	49
7	21	50	67	19	36	27	24
37	30	53	2	27	24	46	32
17	28	49	42	24	35	56	0
13	34	36	63	20	31	1	54
30	29	7	59	26	22	25	53
2	31	51	64	17	30	27	27
36	34	27	30	15	1	50	58
33	15	48	20	35	31	40	29
32	11	38	33	36	31	45	26
33	22	45	17	34	31	50	19
34	20	53	12	34	31	46	20
30	9	38	36	35	30	35	36
30	14	27	46	35	31	32	38
28	23	24	47	32	28	19	51
27	17	16	61	33	29	26	42
31	23	57	8	32	29	46	25
28	17	42	32	34	25	65	9
33	23	52	12	33	31	50	18
33	23	52	11	33	31	49	19
21	18	35	53	30	18	16	61
23	20	12	65	32	27	26	46
25	24	16	60	31	27	16	52
25	23	14	62	31	27	19	49

Table 15: Activations produced by an FSCL$_K$ network employing integer computations and using true robotic sensory values.

Conclusions and Future Work

7

This thesis has examined the theoretical properties and reported experimental results surrounding the use of competitive learning in the unsupervised extraction of visual representations for autonomous mobile robots. The performance of four algorithms were evaluated in the context of both simple two dimensional problems and higher dimensional tasks involving modelled robot vision. Traditionally, neural algorithms are compared on high performance floating point processors with virtually unlimited memory and energy resources, and evaluated according to ultimate errors and perhaps speed of convergence. The situation in our mobile robots (as in many other portable computing situations) is quite different. Here it is necessary to take account of limited precision, limited memory resources and restricted energy (battery) budgets. Which algorithms are most effective under these conditions is established in this thesis for the first time (at least for input-output systems comparable to the current robots).

In order to achieve these results, it was necessary to design and build a suitable mobile robot, to select a reasonable sensory system (based on the jumping spiders which were studied at some length), to design a suitable environmental stimulus system (the PIC controlled light board) and to write a significant piece of software. One example of the significant results achieved through this process was that, using competitive learning algorithms, the robot could properly represent stationary patterns, having been trained only on moving ones.

With regard to the performance of the individual algorithms, standard competitive learning, though computationally efficient, has been shown to possess inherent limitations in its ability to solve the vision based tasks investigated in this thesis. This is primary a result of the algorithm's propensity to orphan units and thereby produce generally poor solutions. The inappropriate allocation of system resources is a direct consequence of the simple winner-take-all nature of the approach, which provides no mechanism to ensure effective use of all units. This situation generally arises in response to poor initialization of the network weights.

In contrast, soft competitive learning was found to produce good solutions to most problems, though it does require careful selection of network parameters in order to achieve these results. Specifically, the quality of the solutions were found to be dependent on the choice of variance used by the radial-basis-function units. Employing a variance decay factor helped to reduce the effects of this problem. The most significant drawback to the use of this algorithm in applications such as mobile robots is its complexity. A large amount of computation is required in the calculation of network activations, owing mainly to the need to compute an exponential function. However, it was discovered that this limitation can be reduced by employing lookup tables in place of the exponential with no significant loss of performance.

As well, the issue of appropriate network size was addressed in the context of SCL and it was discovered that this algorithm was able to effectively distribute the available units over the data, even in the presence of a surplus or shortage of units. As well, it was further suggested that an optimal number of units can be selected by intentionally supplying the network with a surplus of units at the start of training, and later pruning the network in order to achieve the optimal network size for the particular data being clustered.

In addition to standard and soft competitive learning, two versions of frequency sensitive competitive learning (DeSieno and Krishnamurthy) were evaluated and found to produce markedly different results in contrast to the superficial similarity of the two techniques. Through experiment it was demonstrated that both of these algorithms are capable of learning to solve challenging vision based tasks, but that the solutions achieved by $FSCL_K$ are generally superior and require less computation. It was further observed that $FSCL_D$ required careful selection of the bias factor in order to achieve acceptable solutions to these problems, making this algorithm awkward to use in many instances. In fact, in situations where network weights are poorly initialized, the way in which $FSCL_D$ incorporates the frequency dependence as an additive component of the distance calculation makes it incapable of locating useable solutions. It may be possible to correct this shortcoming by introducing a bias factor decay.

Unlike $FSCL_D$, $FSCL_K$ was able to reliably locate good solutions to even the most challenging test problems presented in this thesis, and only produced sub-standard results when using very high learning rates. The fact that $FSCL_K$ includes the frequency dependent property of the learning as a multiplicative component of the distance calculations avoids the difficulties encountered by the $FSCL_D$ approach. Furthermore, the simplicity of the algorithm makes it attractive for use in computationally restricted environments.

Another unique contribution arising from this thesis was the modification of both $FSCL_D$ and $FSCL_K$ to use analog activations in place of the winner-take-all activations present in the original implementations of these algorithms. This conversion from local to distributed representations allowed the networks to more correctly represent the data distributions being modelled and thereby provide better generalization to novel input patterns.

For experiments involving the physical robotic system, the $FSCL_K$ algorithm was selected based on its reliability and modest computational requirements. These tests demonstrated that the algorithm is capable of operating in environments which provide only integer arithmetic and both limited memory and computing resources. It was also found that the robot was able to learn a subset of the vision task presented to it. Its failure to completely solve this task was due to limitations of the robotic sensory apparatus and not that of the learning algorithm itself. This is not a problem with the number of sensors selected for this robot, but rather the properties of the specific devices used (i.e. L14C1 phototransistors). The use of six simple sensors provides a much more complex and less ambiguous representations of visual scenes than using only two. Too few sensors impoverish the representation and too many sensors require a huge expansion of the training data. In interesting problems these inputs are highly correlated and the complexity of the problem grows exponentially rather than linearly in the number of sensors or input dimensionality.

What we believe is new and most valuable here is the realization that the vision of a simple mobile robot with multiple sensors (and hence a large input dimensionality) can be based primarily on competitive learning at all, and that this forms an efficient representation for subsequent (supervised) learning which associates these internal states with motor responses. This approach is quite different than that typically employed with vision in robots.

7.1 Future Work

The robotic vision task presented in this thesis, though challenging, is only a basic first step in the development of adaptive visual system for mobile robotics. While it demonstrates that reasonable adaptive vision is possible, there is still much work which can be done in terms of both the sensory systems themselves and how these are used by the robot. The flexibility of the light board allows for significantly more complicated visual experiences to be investigated. This, coupled with the expansion of the robot's sensory environment through the introduction of additional time-delayed inputs, would permit the investigation of much more complex visual phenomena. Even in the absence of motion on the part of the robot, the richness of the visual experiences that can be generated are immense.

Permitting the robot to move in response to its visual environment extends the range of visual situations greatly. The fact that changes in the observed sensor values are now a result of the physical motion of the robot introduces additional complexities which need to be investigated. The ability of the robot to move has a direct impact on what is seen by the sensors, and this can allow the robot to choose what it looks at based on how it orients itself in the environment. Such behaviour has been observed in spiders and other animals, including humans.

In the context of this thesis the characteristics of the robot's sensors have been assumed to be fixed. However, it is well known that these characteristics can change over time owing to a variety of effects such as temperature or simply aging of components. The adaptability of the neural algorithms makes them capable of compensating for these changing behaviours. However, there are a number of issues relating to these changes which should be examined. Specifically, how quickly can an algorithm reliably track changes in sensor values?

As well, through the course of this thesis it became quite clear that $FSCL_K$ is an extremely robust algorithm. It has demonstrated an ability to solve virtually all problems encountered. In fact, it has performed so well that the bounds of its capabilities remain largely undetermined. It would therefore be instructive to perform further testing on this particular algorithm in order to better identify the extent of its capabilities.

The present work has involved an attempt to improve the design of an artificial creature (for practical applications) by studying the anatomy and behaviour of a simple biological animal. A more ambitious direction for future work on this subject is to reverse this process, and to employ a hardware-software system such as that developed in this thesis to explore the algorithms that may be at work in a real animal. Physiologists know a great deal about the anatomy, and function at the molecular and cell levels, of animals. Similarly, behavioural biologists and zoologists know a great deal about the behaviour of animals in accomplishing their goals of foraging, escaping from predators, courtship and reproduction, grooming and other survival tasks. What is poorly understood is the high-level algorithms which connect the physiology with the observed behaviour. The current robot can be extended in many ways in terms of sensory systems, locomotion, computational precision, amount of available memory, etcetera, and the environment can be extended easily beyond the simple light board apparatus designed in this work. Experiments can be conducted which in an abstract way mimic the behaviour of animals, and these may be connected to comparisons among algorithms for learning, classification, control or other signal processing.

Finally, in addition to the scientific findings, this thesis has resulted in the production of the Claymore neural network simulator. While this software worked well for the simulations conducted, there are many additions which could be introduced to further improve its usability in future studies. These include the addition of new neural algorithms, automated weight tracking, integrated plotting capabilities, and the ability to simulate hybrid networks (networks using different algorithms in different layers).

References

[1] Y. Le Cun, B. Boser J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Computation*, vol. 1, pp. 541–551, 1989.

[2] L. D. Jackel, B. Boser, J. S. Denker, H. P. Graf, Y. Le Cun, I. Guyon, D. Henderson, R. E. Howard, and S. A. Solla, "Handwritten requirements for neural-net optical character recognition," *Proceedings of the International Conference on Neural Networks*, San Diego, CA, vol. II, pp. 855–861, 1990.

[3] D. Pomerleau, "Rapid Adapting Machine Vision for Automated Vehicle Steering," IEEE Expert, vol. 11, no. 2, pp. 19–27, 1996.

[4] T. Gozani, P. Ryge, P. Shea, C. Seher, R. E. Morgado, "Explosive detection system based on thermal neutron activation," *IEEE Aerospace and Electronic Systems Magazine*, vol. 4, no. 12, pp. 17–20, 1989.

[5] G. E. Hinton, *Neural Networks for Industry*, Toronto, 1997.

[6] R. P. Gorman, T. J. Sejnowski, "Analysis of Hidden Units in a Layerd Network Trained to Classify Sonar Targets," *Neural Networks*, vol. 1, pp. 75–89, 1988.

[7] R. P. Gorman, T. J. Sejnowski, "Learned Classification of Sonar Targets Using a Massively-Parallel Network," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, pp. 1135–1140, 1988.

[8] J. Hertz, A. Krogh, R. Palmer, *Introduction to the Theory of Neural Computation*, Redwood City, CA: Addison-Wesley, 1991.

[9] D. E. Rumelhart, G. E. Hinton, R. J. Williams, "Learning representations by backpropagation of errors," *Nature*, vol. 323, pp. 533–536, 1986.

[10] D. E. Rumelhart, D. Zipser, "Feature Discovery by Competitive Learning," *Cognitive Science*, 9, pp. 75–112, 1985.

[11] D. DeSieno, "Adding a Conscience to Competitive Learning," *IEEE International Conference on Neural Networks*, San Diego, Vol. I, pp. 117–124, 1988.

[12] J. Moody, C. J. Darken, "Fast Learning in Networks of Locally-Tuned Processing Units," *Neural Computation*, 1, pp. 281–294, 1989.

[13] T. Kohonen, "Self-Organized Formation of Topologically Correct Feature Maps," *Biological Cybernetics*, Vol. 43, pp. 59–69, 1982.

[14] T. Kohonen, Self-Organizing Maps, New York, NY:Springer-Verlag, 1995.

[15] C. R. Schneider, *Analog CMOS Circuits for Artificial Neural Networks*, Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, University of Manitoba, 1991.

[16] C. R. Schneider, H. C. Card, "Analog CMOS Deterministic Boltzmann Circuits," *IEEE Journal of Solid-State Circuits*, vol. 28, pp. 907–914, August, 1993.

[17] R. S. Schneider, *Deterministic Boltzmann Machines: Learning Instabilities and Hardware Implications*, Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, University of Manitoba, 1993.

[18] B. K. Dolenko, *Performance and Hardware Compatibility of Backpropagation and Cascade Correlation Learning Algorithms*, M.Sc. Thesis, Dept. of Electrical and Computer Engineering, University of Manitoba, 1992.

[19] B. K. Dolenko, H. C. Card, "Tolerance to Analog Hardware of On-Chip Learning in Backpropagation Networks," *IEEE Transactions on Neural Networks*, vol. 6, pp. 1045–1052, September, 1995.
[20] J. A. Dickson, *Stochastic Arithmetic Implementations of Artificial Neural Networks*, M.Sc. Thesis, Dept. of Electrical and Computer Engineering, University of Manitoba, 1992.
[21] R. K. W. Ng, *Rapid-Prototyping of Artificial Neural Networks*, M.Sc. Thesis, Dept. of Electrical and Computer Engineering, University of Manitoba, 1995.
[22] B. Brown, *Soft Competitive Learning using Stochastic Arithmetic*, M.Sc. Thesis, Dept. of Electrical and Computer Engineering, University of Manitoba, 1998.
[23] D. K. McNeill, *Unsupervised Learning in Analog Networks*, M.Sc. Thesis, Dept. of Electrical and Computer Engineering, University of Manitoba, 1993.
[24] H. C. Card, D. K. McNeill, C. R. Schneider, "Analog VLSI Circuits for Competitive Learning Networks," *Journal of Analog Integrated Circuits and Signal Processing*, vol. 13, no. 3, pp. 291–314, 1998.
[25] S. Kamarsu, Neural Code-Excited Linear Prediction for Low Power Adaptive Speech Coding, M.Sc. Thesis, Dept. of Electrical and Computer Engineering, University of Manitoba, 1995.
[26] B. Gilbert, "A High-Performance Monolithic Multiplier Using Active Feedback," *IEEE Journal of Solid-state Circuits*, vol. SC-9, no. 6, pp. 364–373, 1974.
[27] G. K. Rosendahl, *Polymorphic Computing Paradigms Realized for a FPD Based Multicomputer*, Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, University of Manitoba, 1995.
[28] Course Notes for 6.270, *The MIT Lego Robot Design Course*, Dept. of Electrical Engineering and Computer Science, Massachussetts Institute of Technology, 1994.
[29] F. Martin, *Handy Board Specifications*, Media Laboratory, Massachussettes Institute of Technology, 1996.
[30] D. van Camp, T. Plate, G. E. Hinton, *The Xerion Neural Network Simulator*, Department of Computer Science, University of Toronto, 1991.
[31] C. M. Bishop, *Neural Networks for Pattern Recognition*, Oxford University Press, 1995.
[32] A. K. Krishnamurthy, S. C. Ahalt, D. E. Melton, and P. Chen, "Neural Networks for Vector Quantization of Speech and Images," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 8, pp. 1449–1457, 1990.
[33] S. C. Ahalt, A. K. Krishnamurth, P. Chen, and D. E. Mellton, "Competitive Learning Algorithms for Vector Quantization," *Neural Networks*, vol. 3, pp. 277–290, 1990.
[34] T. K. Moon, "The Expectation-Maximization Algorithm," *IEEE Signal Processing Magazine*, vol. 13, no. 6, pp. 47–60, 1996.
[35] R. F. Foelix, Biology of Spiders, Harvard Univesrity Press, 1982.
[36] L. Forster, "Visual Communication in Jumping Spiders," Chapter 5 of *Spider Communication: Mechanisms and Ecological Significance*, P. N. Witt and J. S. Rovner, eds., Princeton University Press, 1982.
[37] R. R. Jackson, "The Behavior of Communicating in Jumping Spiders (Salticidae)," Chapter 6 of *Spider Communication: Mechanisms and Ecological Significance*, P. N. Witt and J. S. Rovner, eds., Princeton University Press, 1982.
[38] J. L. Jones and A. M. Flynn, *Mobile Robots: Inspiration to Implementation*, A K Peters, 1993.
[39] B. Webb, "A Cricket Robot," *Scientific American*, pp. 94–99, Dec. 1996.

Algorithm Source Code

A

A.1 Hard Competitive Learning

A.1.1 HCL Header File (HCL.h)

```
/*******************************************************
 *                                                     *
 *  Claymore -- Hard/Standard Competitive Learning Algorithm  *
 *                                                     *
 *  This file contains the algorithm specific header information *
 *  for the HCL neural network algorithm.              *
 *                                                     *
 *******************************************************/

/* Algorithm Specific Resouce Numbers */
#define  rHCLAlgorithmSettingsDialog  132

enum {  /* Algorithm Settings Menu Item Numbers */
    kHCLAlgorithmEpsilon = 1
    };

struct HCLneuronData    /* NEURON */
  {
  float  totalInput;    /* Sum of all weighted inputs to the neuron */
  };
typedef struct HCLneuronData  HCLneuronData;

struct HCLsynapseData    /* SYNAPSE */
  {
  float  difference;    /* Stores the value (weight-input) */
  };
typedef struct HCLsynapseData  HCLsynapseData;

struct HCLlayerData    /* LAYER */
  {
  neuron *winner;       /* Pointer to the neuron winning the competition */
  };
typedef struct HCLlayerData  HCLlayerData;

struct HCLnetworkData  /* NETWORK */
  {
  float epsilon;        /* Network learning rate */
  };
typedef struct HCLnetworkData  HCLnetworkData;

/*** Prototypes ***/
int HCLinitNetworkData(network *theNetwork);
int HCLinitLayerData(layer *theLayer);
int HCLinitNeuronData(neuron *theNeuron);
int HCLinitSynapseData(synapse *theSynapse);
void HCLcomputeSums(network *net);
void HCLupdateActivations(network *net);
void HCLupdateWeights(network *net);
void HCLdoEpoch(network *net, dataset *theDataset);
void HCLapplyVector(network *net, dataset *theDataset, int vectorNumber);
void HCLassembleStrings(network *net, char *theString);
void HCLSetAlgorithmParameterText(int parameterNumber, unsigned char *theText);
void HCLGetAlgorithmParameterText(int parameterNumber, unsigned char *theText);
void HCLResetAlgorithmParameters();
```

A.1.2 HCL Algorithm Code (HCL.c)

```
/***********************************************************
 *                                                         *
 * Claymore -- Hard/Standard Competitive Learning Algorithm *
 *                                                         *
 * This file contains the algorithm specific routines for the *
 * simulation of the HCL neural network algorithm.  It     *
 * contains no code specific to the construction of the    *
 * network it self. It only provides the functions necessary *
 * for learning.  Network construction routines reside in  *
 * the sim.c module.                                       *
 *                                                         *
 ***********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "macSim.h"
#include "macHCL.h"
#include "macGlobals.h"

/*****************************************************************
 ** HCLinitNetworkData
 **
 ** This function performs any initialization necessary when a
 ** network data record is created. (Algorithm dependent)
 **
 *****************************************************************/

int HCLinitNetworkData(network *theNetwork)
{
  if ((theNetwork->data = malloc(sizeof(HCLnetworkData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initNetworkData");
    return(MALLOC_FAILED);
    }
  ((HCLnetworkData *)theNetwork->data)->epsilon= 0.001;  /* Set an initial learning rate */
  return(0);
}


/*****************************************************************
 ** HCLinitLayerData
 **
 ** This function performs any initialization necessary when a
 ** layer data record is created. (Algorithm dependent)
 **
 *****************************************************************/

int HCLinitLayerData(layer *theLayer)
{
  if ((theLayer->data = malloc(sizeof(HCLlayerData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initLayerData");
    return(MALLOC_FAILED);
    }
  ((HCLlayerData *)theLayer->data)->winner = NULL;
  return(0);
}


/*****************************************************************
 ** HCLinitNeuronData
 **
 ** This function performs any initialization necessary when a
 ** neuron data record is created. (Algorithm dependent)
 **
 *****************************************************************/

int HCLinitNeuronData(neuron *theNeuron)
{
  if ((theNeuron->data = malloc(sizeof(HCLneuronData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initNeuronData");
    return(MALLOC_FAILED);
    }
  ((HCLneuronData *)theNeuron->data)->totalInput = 0.0;
  return(0);
}


/*****************************************************************
 ** HCLinitSynapseData
 **
 ** This function performs any initialization necessary when a
 ** synapse data record is created. (Algorithm dependent)
 **
 *****************************************************************/

int HCLinitSynapseData(synapse *theSynapse)
{
  if ((theSynapse->data = malloc(sizeof(HCLsynapseData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initSynapseData");
    return(MALLOC_FAILED);
    }
  return(0);
}


/*****************************************************************
 ** HCLcomputeSums
 **
 ** This function does the forward propagation through a network,
 ** computing the total input for each unit and determining the winner
 ** for each layer.  This is pass 1 of the network computations.  It
 ** returns the network error resulting from that operation.
 **
 *****************************************************************/

void HCLcomputeSums(network *net)
{
  layer   *layerPtr;
  neuron  *neuronPtr, *winnerPtr;
  synapse *synapsePtr;
  float   totalInput;

  layerPtr = net->layers;
  while (layerPtr != NULL)    /* Traverse the layers */
    {
    winnerPtr = neuronPtr = layerPtr->neurons;
    while (neuronPtr != NULL) /* Traverse the neurons */
      {
      if (!(neuronPtr->lock))
        {
        totalInput = 0.0;
        synapsePtr = neuronPtr->synapseIn;
        while (synapsePtr != NULL)
          {
          ((HCLsynapseData *)synapsePtr->data)->difference =
                  synapsePtr->weight - synapsePtr->neuronIn->activation;
          totalInput += square(((HCLsynapseData *)synapsePtr->data)->difference);
          synapsePtr = synapsePtr->nextIn;
          }
        ((HCLneuronData *)neuronPtr->data)->totalInput = totalInput;
        if (((HCLneuronData *)winnerPtr->data)->totalInput > totalInput)
          winnerPtr = neuronPtr;
        }
      neuronPtr = neuronPtr->next;
      }
    net->error += ((HCLneuronData *)winnerPtr->data)->totalInput;
    ((HCLlayerData *)layerPtr->data)->winner = winnerPtr;
    layerPtr = layerPtr->next;
    }
}


/*****************************************************************
 ** HCLupdateActivations
 **
 ** This function traverses the network and sets the activations of all
 ** neurons to the appropriate values.  It is called after computeSums.
 ** This is pass 2 of the network computations.
 **
 *****************************************************************/
```

```
void HCLupdateActivations(network *net)
{
  layer   *layerPtr;
  neuron  *neuronPtr;

  layerPtr = net->layers->next;  /* Skip the first layer since it is the input layer */
  while (layerPtr != NULL)     /* Traverse layers */
    {
    neuronPtr = layerPtr->neurons;
    while (neuronPtr != NULL)  /* Traverse neurons */
      {
      neuronPtr->activation = 0.0;  /* Clear old activations */
      neuronPtr = neuronPtr->next;
      }
    ((HCLlayerData *)layerPtr->data)->winner->activation = 1.0;  /* Set winner's activation */
    layerPtr = layerPtr->next;
    }
}


/****************************************************************
 ** HCLupdateWeights
 **
 ** This function updates the weights for the winning unit in each layer.
 **
 ****************************************************************/

void HCLupdateWeights(network *net)
{
  layer   *layerPtr;
  synapse *synapsePtr;

  layerPtr = net->layers->next;
  while (layerPtr != NULL)      /* Traverse layers */
    {
    synapsePtr = ((HCLlayerData *)layerPtr->data)->winner->synapseIn;
    while (synapsePtr != NULL)  /* Traverse synapses of winner */
      {
      synapsePtr->weight -= ((HCLnetworkData *)net->data)->epsilon * ((HCLsynapseData *)synapsePtr-
>data)->difference;
      synapsePtr = synapsePtr->nextIn;
      }
    layerPtr = layerPtr->next;
    }
}


/****************************************************************
 ** HCLdoEpoch
 **
 ** Applies the set of input patterns to the network and calls
 ** computeSums, updateActivations and updateWeights to perform the
 ** learning.
 **
 ****************************************************************/

void HCLdoEpoch(network *net, dataset *theDataset)
{
  int          index;
  neuron       *theNeuron;
  dataElement *element;

  /* Set pointer to the first input vector within the dataset. */
  element = theDataset->data;
  net->error = 0.0;           /* Clear the network error. */
  while(element != NULL)      /* Apply each input vector in turn. */
    {
    index = 0;
    theNeuron = net->layers->neurons;
    while(theNeuron != NULL)                             /* Set layer 0  */
      {                                                  /* activations  */
      theNeuron->activation = element->inputData[index++]; /* equal to the */
      theNeuron = theNeuron->next;                       /* the input    */
      }                                                  /* vector.      */
    element = element->next;

    HCLcomputeSums(net);          /* Computed the weighted sums for each neuron. */
    HCLupdateActivations(net);    /* Update all neuron activations at once. */
    if (!theNet->batch)        /* If weights are to be updated in batch mode, */
      HCLupdateWeights(net);      /* then don't do it here. */
    }
  if (theNet->batch)           /* Update weights here when in batch mode. */
```

```
    HCLupdateWeights(net);
}


/*****************************************************************
 ** HCLapplyVector
 **
 ** Applies the specified vector number to the network and calls HCLcomputeSums and
 ** HCLupdateActivations to actually perform the computations.
 **
 *****************************************************************/

void HCLapplyVector(network *net, dataset *theDataset, int vectorNumber)
{
    int         index;
    neuron      *theNeuron;
    dataElement *element;

    /* Set pointer to the first input vector within the dataset. */
    element = theDataset->data;
    index = 0;
    while(index++ != vectorNumber)       /* Locate the specified  input vector. */
        element = element->next;
    theNeuron = net->layers->neurons;
    index = 0;
    while(theNeuron != NULL)                                  /* Set layer 0  */
        {                                                     /* activations  */
        theNeuron->activation = element->inputData[index++]; /* equal to the */
        theNeuron = theNeuron->next;                          /* the input    */
        }                                                     /* vector.      */
    HCLcomputeSums(net);         /* Compute the weighted sums for each neuron. */
    HCLupdateActivations(net);   /* Update all neuron activations at once. */
}


/*****************************************************************
 ** HCLassembleStrings
 **
 ** Prints the algorithm specific variables to a string and returns that string for display.
 **
 *****************************************************************/

void HCLassembleStrings(network *net, char *theString)
{
    if (net != NULL)
        sprintf(theString, "Hard (Standard) CL\r\rLearning Rate (epsilon) = %f\r",
                ((HCLnetworkData *)net->data)->epsilon);
}


/*****************************************************************
 ** HCLSetAlgorithmParameterText
 **
 ** Prints the algorithm specific variables to a string and returns that string for display.
 **
 *****************************************************************/

void HCLSetAlgorithmParameterText(int parameterNumber, unsigned char *theText)
{
    char *theCText;

    theCText = p2cstr(theText);
    switch (parameterNumber)
        {
        case kHCLAlgorithmEpsilon :
            ((HCLnetworkData *)theNet->data)->epsilon = atof(theCText);
            break;
        }
}


/*****************************************************************
 ** HCLGetAlgorithmParameterText
 **
 ** Prints the algorithm specific variables to a string and returns that string for display.
 **
 *****************************************************************/

void HCLGetAlgorithmParameterText(int parameterNumber, unsigned char *theText)
{
    switch (parameterNumber)
        {
```

```
        case kHCLAlgorithmEpsilon :
            sprintf((char *)theText, "%f", ((HCLnetworkData *)theNet->data)->epsilon);
            break;
        }
    c2pstr((char *)theText);
}

/******************************************************************
 ** HCLResetAlgorithmParameters
 **
 ** Resets any network parameters so that learning operates in the same way it would
 ** have if the network was deleted and an identical network constructed.
 **
 ******************************************************************/

void HCLResetAlgorithmParameters()
{
/* For this algorithm we have nothing to do here. */
}
```

A.2 DeSieno Frequency Sensitive Competitive Learning

A.2.1 FSCL$_D$ Header File (FSCLD.h)

```
/************************************************************
 *
 *  Claymore -- Frequency Sensitive Competitive Learning Algorithm
 *
 *  This file contains the algorithm specific header information
 *  for the FSCL neural network algorithm.
 *
 ************************************************************/
/* Algorithm Specific Resouce Numbers */
#define  rFSCLAlgorithmSettingsDialog  130

enum {  /* Algorithm Settings Menu Item Numbers */
    kFSCLAlgorithmEpsilon = 1,
    kFSCLAlgorithmConscience = 2,
    kFSCLAlgorithmBias = 3
    };

struct FSCLneuronData     /* NEURON */
  {
  float  totalInput;     /* Sum of all weighted inputs to the neuron */
  float  winningProportion;  /* Fraction of time the neuron wins a competition */
  float  biasedTotalInput;  /* Sum of all weighted inputs minus bias term for this neuron */
  };
typedef struct FSCLneuronData  FSCLneuronData;

struct FSCLsynapseData     /* SYNAPSE */
  {
  float  difference;     /* Stores the value (weight-input) */
  };
typedef struct FSCLsynapseData  FSCLsynapseData;

struct FSCLlayerData     /* LAYER */
  {
  neuron *winner;        /* Pointer to the neuron winning the competition independet of conscience */
  neuron *updateWinner;  /* Pointer to neuron winning competition under influence of conscience */
  int  numNeurons;  /* A count of the number of neurons in the layer */
  float  normFactor;  /* Weight normalization value */
  };
typedef struct FSCLlayerData  FSCLlayerData;

struct FSCLnetworkData  /* NETWORK */
  {
  float  epsilon;         /* Network learning rate */
  float  proportionAdjustment;  /* Constant determining relative strength of conscience */
  float  biasFactor;  /* Constant to determine bias strength */
  };
  typedef struct FSCLnetworkData  FSCLnetworkData;

/*** Prototypes ***/
int FSCLinitNetworkData(network *theNetwork);
int FSCLinitLayerData(layer *theLayer);
int FSCLinitNeuronData(neuron *theNeuron);
int FSCLinitSynapseData(synapse *theSynapse);
```

```
void FSCLcomputeSums(network *net);
void FSCLupdateActivations(network *net);
void FSCLupdateWeights(network *net);
void FSCLdoEpoch(network *net, dataset *theDataset);
void FSCLapplyVector(network *net, dataset *theDataset, int vectorNumber);
void FSCLassembleStrings(network *net, char *theString);
void FSCLGetAlgorithmParameterText(int parameterNumber, unsigned char *theText);
void FSCLSetAlgorithmParameterText(int parameterNumber, unsigned char *theText);
void FSCLResetAlgorithmParameters();
```

A.2.2 FSCL$_D$ Algorithm Code (FSCLD.c)

```
/************************************************************
 *
 *  Claymore -- Frequency Sensitive Competitive Learning Algorithm
 *
 *  This file contains the algorithm specific routines for the
 *  simulation of the FSCL neural network algorithm.  It
 *  contains no code specific to the construction of the
 *  network it self. It only provides the functions necessary
 *  for learning.  Network construction routines reside in
 *  the sim.c module.
 *
 ************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "macSim.h"
#include "macFSCL.h"
#include "macGlobals.h"

/******************************************************************
 ** FSCLinitNetworkData
 **
 ** This function performs any initialization necessary when a
 ** network data record is created. (Algorithm dependent)
 **
 ******************************************************************/

int FSCLinitNetworkData(network *theNetwork)
{
  if ((theNetwork->data = malloc(sizeof(FSCLnetworkData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initNetworkData");
    return(MALLOC_FAILED);
    }
  ((FSCLnetworkData *)theNetwork->data)->epsilon= 0.001;  /* Set an initial learning rate */
  ((FSCLnetworkData *)theNetwork->data)->proportionAdjustment = 0.0001;
  ((FSCLnetworkData *)theNetwork->data)->biasFactor = 10.0;
  return(0);
}


/******************************************************************
 ** FSCLinitLayerData
 **
 ** This function performs any initialization necessary when a
 ** layer data record is created. (Algorithm dependent)
 **
 ******************************************************************/

int FSCLinitLayerData(layer *theLayer)
{
  if ((theLayer->data = malloc(sizeof(FSCLlayerData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initLayerData");
    return(MALLOC_FAILED);
    }
  ((FSCLlayerData *)theLayer->data)->winner = NULL;
  ((FSCLlayerData *)theLayer->data)->updateWinner = NULL;
  ((FSCLlayerData *)theLayer->data)->numNeurons = 0;
  return(0);
}


/******************************************************************
 ** FSCLinitNeuronData
 **
 ** This function performs any initialization necessary when a
 ** neuron data record is created. (Algorithm dependent)
 ***
 *****************************************************************/

int FSCLinitNeuronData(neuron *theNeuron)
{
  if ((theNeuron->data = malloc(sizeof(FSCLneuronData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initNeuronData");
    return(MALLOC_FAILED);
    }
  ((FSCLneuronData *)theNeuron->data)->totalInput = 0.0;
  ((FSCLneuronData *)theNeuron->data)->biasedTotalInput = 0.0;
  ((FSCLneuronData *)theNeuron->data)->winningProportion = 0.0;
  ((FSCLlayerData *)theNeuron->layerPtr->data)->numNeurons++;
  return(0);
}


/*****************************************************************
 ** FSCLinitSynapseData
 **
 ** This function performs any initialization necessary when a
 ** synapse data record is created. (Algorithm dependent)
 **
 *****************************************************************/

int FSCLinitSynapseData(synapse *theSynapse)
{
  if ((theSynapse->data = malloc(sizeof(FSCLsynapseData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initSynapseData");
    return(MALLOC_FAILED);
    }
  return(0);
}


/*****************************************************************
 ** FSCLcomputeSums
 **
 ** This function does the forward propagation through a network,
 ** computing the total input for each unit and determining the winner
 ** for each layer.  This is pass 1 of the network computations.  It
 ** returns the network error resulting from that operation.
 **
 *****************************************************************/

void FSCLcomputeSums(network *net)
{
  layer   *layerPtr;
  neuron  *neuronPtr, *winnerPtr, *updateWinnerPtr;
  synapse *synapsePtr;
  float   totalInput;

  layerPtr = net->layers->next;
  while (layerPtr != NULL)    /* Traverse the layers */
    {
    winnerPtr = updateWinnerPtr = neuronPtr = layerPtr->neurons;
    ((FSCLlayerData *)layerPtr->data)->normFactor = 0.0;    /* Clear the normalization value */
    while (neuronPtr != NULL) /* Traverse the neurons */
      {
      if (!(neuronPtr->lock))
        {
        totalInput = 0.0;
        synapsePtr = neuronPtr->synapseIn;
        while (synapsePtr != NULL)
          {
          ((FSCLsynapseData *)synapsePtr->data)->difference =
                      synapsePtr->weight - synapsePtr->neuronIn->activation;
          totalInput += square(((FSCLsynapseData *)synapsePtr->data)->difference);
          synapsePtr = synapsePtr->nextIn;
          }
        ((FSCLneuronData *)neuronPtr->data)->totalInput = totalInput;
       ((FSCLneuronData *)neuronPtr->data)->biasedTotalInput = totalInput - ((FSCLnetworkData *)net-
>data)->biasFactor *
            ((1.0 / ((FSCLlayerData *)layerPtr->data)->numNeurons) - ((FSCLneuronData *)neuronPtr-
>data)->winningProportion);
        if (((FSCLneuronData *)winnerPtr->data)->totalInput > totalInput)
          winnerPtr = neuronPtr;
      if (((FSCLneuronData *)updateWinnerPtr->data)->biasedTotalInput > ((FSCLneuronData *)neuronPtr-
>data)->biasedTotalInput)
          updateWinnerPtr = neuronPtr;
        ((FSCLlayerData *)layerPtr->data)->normFactor += 1.0 / totalInput;
        }
      neuronPtr = neuronPtr->next;
      }
    net->error += ((FSCLneuronData *)winnerPtr->data)->totalInput;
    ((FSCLlayerData *)layerPtr->data)->winner = winnerPtr;
    ((FSCLlayerData *)layerPtr->data)->updateWinner = updateWinnerPtr;
    layerPtr = layerPtr->next;
    }
}


/****************************************************************
 ** FSCLupdateActivations
 **
 ** This function traverses the network and sets the activations of all
 ** neurons to the appropriate values.  It is called after FSCLcomputeSums.
 ** This is pass 2 of the network computations.
 **
 ****************************************************************/

void FSCLupdateActivations(network *net)
{
  layer   *layerPtr;
  neuron  *neuronPtr;
  float   winnerProportion;

  layerPtr = net->layers->next;  /* Skip the first layer since it is the input layer */
  while (layerPtr != NULL)      /* Traverse layers */
    {
    neuronPtr = layerPtr->neurons;
    winnerProportion = ((FSCLneuronData *)((FSCLlayerData *)layerPtr->data)->winner->data)-
>winningProportion; /* Save winners proportion value for later */
    while (neuronPtr != NULL)  /* Traverse neurons */
      {
      /* neuronPtr->activation = 0.0;  /* Clear old activations */
      neuronPtr->activation =  (1.0 / ((FSCLneuronData *)neuronPtr->data)->totalInput) /
((FSCLlayerData *)layerPtr->data)->normFactor;
      ((FSCLneuronData *)neuronPtr->data)->winningProportion -=
         ((FSCLnetworkData *)net->data)->proportionAdjustment * ((FSCLneuronData *)neuronPtr->data)-
>winningProportion;
      neuronPtr = neuronPtr->next;
      }
    /* ((FSCLlayerData *)layerPtr->data)->winner->activation = 1.0;  /* Set winner's activation */
    ((FSCLneuronData *)((FSCLlayerData *)layerPtr->data)->winner->data)->winningProportion =
       winnerProportion + ((FSCLnetworkData *)net->data)->proportionAdjustment * (1.0 -
winnerProportion);
    layerPtr = layerPtr->next;
    }
}


/****************************************************************
 ** FSCLupdateWeights
 **
 ** This function updates the weights for the winning unit in each layer.
 **
 ****************************************************************/

void FSCLupdateWeights(network *net)
{
  layer   *layerPtr;
  synapse *synapsePtr;

  layerPtr = net->layers->next;
  while (layerPtr != NULL)      /* Traverse layers */
    {
    synapsePtr = ((FSCLlayerData *)layerPtr->data)->updateWinner->synapseIn;
    while (synapsePtr != NULL)  /* Traverse synapses of winner */
      {
      synapsePtr->weight -= ((FSCLnetworkData *)net->data)->epsilon * ((FSCLsynapseData *)synapsePtr-
>data)->difference;
      synapsePtr = synapsePtr->nextIn;
      }
    layerPtr = layerPtr->next;
    }
}


/****************************************************************
 ** FSCLdoEpoch
 **
 ** Applies the set of input patterns to the network and calls FSCLcomputeSums.
 ** FSCLupdateActivations and FSCLupdateWeights to perform the learning.
 **
 *****************************************************************/

void FSCLdoEpoch(network *net, dataset *theDataset)
{
  int         index;
  neuron      *theNeuron;
  dataElement *element;

  /* Set pointer to the first input vector within the dataset. */
  element = theDataset->data;
  net->error = 0.0;            /* Clear the network error. */
  while(element != NULL)       /* Apply each input vector in turn. */
    {
    index = 0;
    theNeuron = net->layers->neurons;
    while(theNeuron != NULL)                            /* Set layer 0  */
      {                                                 /* activations  */
      theNeuron->activation = element->inputData[index++]; /* equal to the */
      theNeuron = theNeuron->next;                      /* the input    */
      }                                                 /* vector.      */
    element = element->next;

    FSCLcomputeSums(net);         /* Computed the weighted sums for each neuron. */
    FSCLupdateActivations(net);   /* Update all neuron activations at once. */
    if (!theNet->batch)        /* If weights are to be updated in batch mode, */
      FSCLupdateWeights(net);     /* then don't do it here. */
    }
  if (theNet->batch)           /* Update weights here when in batch mode. */
    FSCLupdateWeights(net);
}


/****************************************************************
 ** FSCLapplyVector
 **
 ** Applies the specified vector number to the network and calls FSCLcomputeSums and
 ** FSCLupdateActivations to actually perform the computations.
 **
 *****************************************************************/

void FSCLapplyVector(network *net, dataset *theDataset, int vectorNumber)
{
    int         index;
    neuron      *theNeuron;
    dataElement *element;

    /* Set pointer to the first input vector within the dataset. */
    element = theDataset->data;
    index = 0;
    while(index++ != vectorNumber)      /* Locate the specified  input vector. */
        element = element->next;
    theNeuron = net->layers->neurons;
    index = 0;
    while(theNeuron != NULL)                              /* Set layer 0  */
        {                                                 /* activations  */
        theNeuron->activation = element->inputData[index++]; /* equal to the */
        theNeuron = theNeuron->next;                      /* the input    */
        }                                                 /* vector.      */
    FSCLcomputeSums(net);         /* Computed the weighted sums for each neuron. */
    FSCLupdateActivations(net);   /* Update all neuron activations at once. */
}


/****************************************************************
 ** FSCLassembleStrings
 **
 ** Prints the algorithm specific variables to a string and returns that string for display.
 **
 *****************************************************************/

void FSCLassembleStrings(network *net, char *theString)
{
    if (net != NULL)
        sprintf(theString, "Frequency Sensitive CL (DeSieno)\r\rLearning Rate (epsilon) = %f\rWinner
Proportion Factor (B) = %f\rBias Factor (C) = %f\r",
            ((FSCLnetworkData *)net->data)->epsilon, ((FSCLnetworkData *)net->data)-
>proportionAdjustment,
            ((FSCLnetworkData *)net->data)->biasFactor);
}
```

```
/*****************************************************************
 ** FSCLSetAlgorithmParameterText
 **
 ** Prints the algorithm specific variables to a string and returns that string for display.
 **
 *****************************************************************/

void FSCLSetAlgorithmParameterText(int parameterNumber, unsigned char *theText)
{
    char *theCText;

    theCText = p2cstr(theText);
    switch (parameterNumber)
        {
        case kFSCLAlgorithmEpsilon :
            ((FSCLnetworkData *)theNet->data)->epsilon = atof(theCText);
            break;
        case kFSCLAlgorithmConscience :
            ((FSCLnetworkData *)theNet->data)->proportionAdjustment = atof(theCText);
            break;
        case kFSCLAlgorithmBias :
            ((FSCLnetworkData *)theNet->data)->biasFactor = atof(theCText);
            break;
        }
}

/*****************************************************************
 ** FSCLGetAlgorithmParameterText
 **
 ** Prints the algorithm specific variables to a string and returns that string for display.
 **
 *****************************************************************/

void FSCLGetAlgorithmParameterText(int parameterNumber, unsigned char *theText)
{
    switch (parameterNumber)
        {
        case kFSCLAlgorithmEpsilon :
            sprintf((char *)theText, "%f", ((FSCLnetworkData *)theNet->data)->epsilon);
            break;
        case kFSCLAlgorithmConscience :
            sprintf((char *)theText, "%f", ((FSCLnetworkData *)theNet->data)->proportionAdjustment);
            break;
        case kFSCLAlgorithmBias :
            sprintf((char *)theText, "%f", ((FSCLnetworkData *)theNet->data)->biasFactor);
            break;
        }
    c2pstr((char *)theText);
}

/*****************************************************************
 ** FSCLResetAlgorithmParameters
 **
 ** Resets any network parameters so that learning operates in the same way it would
 ** have if the network was deleted and an identical network constructed.
 **
 *****************************************************************/

void FSCLResetAlgorithmParameters()
{
/* For this algorithm we have nothing to do here. */
}
```

A.3 Krishnamurthy Frequency Sensitive Competitive Learning

A.3.1 FSCL$_K$ Header File (FSCLK.h)

```
/*********************************************************
 *
 * Claymore -- Frequency Sensitive Competitive Learning Algorithm (Krishnamurthy)
 *
 * This file contains the algorithm specific header information
 * for the FSCL neural network algorithm.
 *
 *********************************************************/
```

```
/* Algorithm Specific Resouce Numbers */
#define  rFSCLKAlgorithmSettingsDialog  135

enum {  /* Algorithm Settings Menu Item Numbers */
    kFSCLKAlgorithmEpsilon = 1,
    kFSCLKAlgorithmFairness = 2
    };

struct FSCLKneuronData    /* NEURON */
  {
  float  totalInput;     /* Sum of all weighted inputs to the neuron */
  float  fairTotalInput;   /* totalInput * fairness function */
  int  winningCount;  /* Number of time the neuron wins a competition */
  };
typedef struct FSCLKneuronData  FSCLKneuronData;

struct FSCLKsynapseData    /* SYNAPSE */
  {
  float  difference;    /* Stores the value (weight-input) */
  };
typedef struct FSCLKsynapseData  FSCLKsynapseData;

struct FSCLKlayerData    /* LAYER */
  {
  neuron *winner;       /* Pointer to the neuron winning the competition independet of conscience */
  int  numNeurons;  /* A count of the number of neurons in the layer */
  float  normFactor;  /* Weight normalization value */
  };
typedef struct FSCLKlayerData  FSCLKlayerData;

struct FSCLKnetworkData  /* NETWORK */
  {
  float  epsilon;         /* Network learning rate */
  float  fairnessFactor;  /* Constant determining relative strength of conscience */
  };
  typedef struct FSCLKnetworkData  FSCLKnetworkData;

/*** Prototypes ***/
int FSCLKinitNetworkData(network *theNetwork);
int FSCLKinitLayerData(layer *theLayer);
int FSCLKinitNeuronData(neuron *theNeuron);
int FSCLKinitSynapseData(synapse *theSynapse);
void FSCLKcomputeSums(network *net);
void FSCLKupdateActivations(network *net);
void FSCLKupdateWeights(network *net);
void FSCLKdoEpoch(network *net, dataset *theDataset);
void FSCLKapplyVector(network *net, dataset *theDataset, int vectorNumber);
void FSCLKassembleStrings(network *net, char *theString);
void FSCLKGetAlgorithmParameterText(int parameterNumber, unsigned char *theText);
void FSCLKSetAlgorithmParameterText(int parameterNumber, unsigned char *theText);
void FSCLKResetAlgorithmParameters();
```

A.3.2 FSCL$_K$ Algorithm Code (FSCLK.c)

```
/**********************************************************
 *
 *  Claymore -- Frequency Sensitive Competitive Learning Algorithm (Krishnamurthy)
 *
 *  This file contains the algorithm specific routines for the
 *  simulation of the FSCL neural network algorithm.  It
 *  contains no code specific to the construction of the
 *  network it self. It only provides the functions necessary
 *  for learning.  Network construction routines reside in
 *  the sim.c module.
 *
 **********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "macSim.h"
#include "macFSCLK.h"
#include "macGlobals.h"

/*****************************************************************
 ** FSCLKinitNetworkData
 **
 ** This function performs any initialization necessary when a
 ** network data record is created. (Algorithm dependent)
 **
 *****************************************************************/
```

```
int FSCLKinitNetworkData(network *theNetwork)
{
  if ((theNetwork->data = malloc(sizeof(FSCLKnetworkData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initNetworkData");
    return(MALLOC_FAILED);
    }
  ((FSCLKnetworkData *)theNetwork->data)->epsilon= 0.001;  /* Set an initial learning rate */
  ((FSCLKnetworkData *)theNetwork->data)->fairnessFactor = 1.0;
  return(0);
}


/*****************************************************************
 ** FSCLKinitLayerData
 **
 ** This function performs any initialization necessary when a
 ** layer data record is created. (Algorithm dependent)
 **
 *****************************************************************/

int FSCLKinitLayerData(layer *theLayer)
{
  if ((theLayer->data = malloc(sizeof(FSCLKlayerData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initLayerData");
    return(MALLOC_FAILED);
    }
  ((FSCLKlayerData *)theLayer->data)->winner = NULL;
  ((FSCLKlayerData *)theLayer->data)->numNeurons = 0;
  return(0);
}


/*****************************************************************
 ** FSCLKinitNeuronData
 **
 ** This function performs any initialization necessary when a
 ** neuron data record is created. (Algorithm dependent)
 **
 *****************************************************************/

int FSCLKinitNeuronData(neuron *theNeuron)
{
  if ((theNeuron->data = malloc(sizeof(FSCLKneuronData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initNeuronData");
    return(MALLOC_FAILED);
    }
  ((FSCLKneuronData *)theNeuron->data)->totalInput = 0.0;
  ((FSCLKneuronData *)theNeuron->data)->fairTotalInput = 0.0;
  ((FSCLKneuronData *)theNeuron->data)->winningCount = 0;
  ((FSCLKlayerData *)theNeuron->layerPtr->data)->numNeurons++;
  return(0);
}


/*****************************************************************
 ** FSCLKinitSynapseData
 **
 ** This function performs any initialization necessary when a
 ** synapse data record is created. (Algorithm dependent)
 **
 *****************************************************************/

int FSCLKinitSynapseData(synapse *theSynapse)
{
  if ((theSynapse->data = malloc(sizeof(FSCLKsynapseData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initSynapseData");
    return(MALLOC_FAILED);
    }
  return(0);
}


/*****************************************************************
 ** FSCLKcomputeSums
 **
 ** This function does the forward propagation through a network,
 ** computing the total input for each unit and determining the winner
 ** for each layer.  This is pass 1 of the network computations.  It
 ** returns the network error resulting from that operation.
 **
 ******************************************************************/

void FSCLKcomputeSums(network *net)
{
  layer   *layerPtr;
  neuron  *neuronPtr, *winnerPtr;
  synapse *synapsePtr;
  float   totalInput, fairness;

  layerPtr = net->layers->next;
  while (layerPtr != NULL)     /* Traverse the layers */
    {
    winnerPtr = neuronPtr = layerPtr->neurons;
    ((FSCLKlayerData *)layerPtr->data)->normFactor = 0.0;     /* Clear the normalization value */
    while (neuronPtr != NULL) /* Traverse the neurons */
      {
      if (!(neuronPtr->lock))
        {
        totalInput = 0.0;
        synapsePtr = neuronPtr->synapseIn;
        while (synapsePtr != NULL)
          {
          ((FSCLKsynapseData *)synapsePtr->data)->difference =
                      synapsePtr->weight - synapsePtr->neuronIn->activation;
          totalInput += square(((FSCLKsynapseData *)synapsePtr->data)->difference);
          synapsePtr = synapsePtr->nextIn;
          }
        ((FSCLKneuronData *)neuronPtr->data)->totalInput = totalInput;
        fairness = ((FSCLKnetworkData *)net->data)->fairnessFactor * (float)((FSCLKneuronData
*)neuronPtr->data)->winningCount;
        ((FSCLKneuronData *)neuronPtr->data)->fairTotalInput = fairness * totalInput;
        if (((FSCLKneuronData *)winnerPtr->data)->fairTotalInput > ((FSCLKneuronData *)neuronPtr-
>data)->fairTotalInput)
          winnerPtr = neuronPtr;
        ((FSCLKlayerData *)layerPtr->data)->normFactor += 1.0 / totalInput;
        }
      neuronPtr = neuronPtr->next;
      }
    net->error += ((FSCLKneuronData *)winnerPtr->data)->totalInput;
    ((FSCLKlayerData *)layerPtr->data)->winner = winnerPtr;
    layerPtr = layerPtr->next;
    }
}


/******************************************************************
 ** FSCLKupdateActivations
 **
 ** This function traverses the network and sets the activations of all
 ** neurons to the appropriate values.  It is called after FSCLKcomputeSums.
 ** This is pass 2 of the network computations.
 **
 ******************************************************************/

void FSCLKupdateActivations(network *net)
{
  layer   *layerPtr;
  neuron  *neuronPtr;

  layerPtr = net->layers->next;  /* Skip the first layer since it is the input layer */
  while (layerPtr != NULL)     /* Traverse layers */
    {
    neuronPtr = layerPtr->neurons;
    while (neuronPtr != NULL)  /* Traverse neurons */
      {
      /* neuronPtr->activation = 0.0;  /* Clear old activations */
      neuronPtr->activation =  (1.0 / ((FSCLKneuronData *)neuronPtr->data)->totalInput) /
((FSCLKlayerData *)layerPtr->data)->normFactor;
      neuronPtr = neuronPtr->next;
      }
    /* ((FSCLKlayerData *)layerPtr->data)->winner->activation = 1.0;  /* Set winner's activation */
    layerPtr = layerPtr->next;
    }
}


/******************************************************************
 ** FSCLKupdateWeights
 **
 ** This function updates the weights for the winning unit in each layer.
 **
 ****************************************************************/

void FSCLKupdateWeights(network *net)
{
  layer   *layerPtr;
  synapse *synapsePtr;

  layerPtr = net->layers->next;
  while (layerPtr != NULL)      /* Traverse layers */
    {
    synapsePtr = ((FSCLKlayerData *)layerPtr->data)->winner->synapseIn;
    ((FSCLKneuronData *)((FSCLKlayerData *)layerPtr->data)->winner->data)->winningCount++;
    while (synapsePtr != NULL)  /* Traverse synapses of winner */
      {
      synapsePtr->weight -= ((FSCLKnetworkData *)net->data)->epsilon * ((FSCLKsynapseData
*)synapsePtr->data)->difference;
      synapsePtr = synapsePtr->nextIn;
      }
    layerPtr = layerPtr->next;
    }
}


/****************************************************************
 ** FSCLKdoEpoch
 **
 ** Applies the set of input patterns to the network and calls FSCLKcomputeSums.
 ** FSCLKupdateActivations and FSCLKupdateWeights to perform the learning.
 **
 ****************************************************************/

void FSCLKdoEpoch(network *net, dataset *theDataset)
{
  int         index;
  neuron      *theNeuron;
  dataElement *element;

  /* Set pointer to the first input vector within the dataset. */
  element = theDataset->data;
  net->error = 0.0;             /* Clear the network error. */
  while(element != NULL)        /* Apply each input vector in turn. */
    {
    index = 0;
    theNeuron = net->layers->neurons;
    while(theNeuron != NULL)                              /* Set layer 0  */
      {                                                   /* activations  */
      theNeuron->activation = element->inputData[index++]; /* equal to the */
      theNeuron = theNeuron->next;                         /* the input    */
      }                                                   /* vector.      */
    element = element->next;

    FSCLKcomputeSums(net);          /* Computed the weighted sums for each neuron. */
    FSCLKupdateActivations(net);    /* Update all neuron activations at once. */
    if (!theNet->batch)         /* If weights are to be updated in batch mode. */
      FSCLKupdateWeights(net);      /* then don't do it here. */
    }
  if (theNet->batch)            /* Update weights here when in batch mode. */
    FSCLKupdateWeights(net);
}


/****************************************************************
 ** FSCLKapplyVector
 **
 ** Applies the specified vector number to the network and calls FSCLKcomputeSums and
 ** FSCLKupdateActivations to actually perform the computations.
 **
 ****************************************************************/

void FSCLKapplyVector(network *net, dataset *theDataset, int vectorNumber)
{
    int         index;
    neuron      *theNeuron;
    dataElement *element;

    /* Set pointer to the first input vector within the dataset. */
    element = theDataset->data;
    index = 0;
    while(index++ != vectorNumber)      /* Locate the specified  input vector. */
        element = element->next;
    theNeuron = net->layers->neurons;
    index = 0;
    while(theNeuron != NULL)                                    /* Set layer 0  */
        {                                                       /* activations  */
        theNeuron->activation = element->inputData[index++]; /* equal to the */
        theNeuron = theNeuron->next;                            /* the input    */
        }                                                       /* vector.      */
    FSCLKcomputeSums(net);          /* Computed the weighted sums for each neuron. */
    FSCLKupdateActivations(net);    /* Update all neuron activations at once. */
}


/*******************************************************************
 ** FSCLKassembleStrings
 **
 ** Prints the algorithm specific variables to a string and returns that string for display.
 **
 *******************************************************************/

void FSCLKassembleStrings(network *net, char *theString)
{
    if (net != NULL)
       sprintf(theString, "Frequency Sensitive CL (Krish.)\r\rLearning Rate (epsilon) = %f\rFairness
Factor = %f\r",
              ((FSCLKnetworkData *)net->data)->epsilon, ((FSCLKnetworkData *)net->data)-
>fairnessFactor);
}


/*******************************************************************
 ** FSCLKSetAlgorithmParameterText
 **
 ** Prints the algorithm specific variables to a string and returns that string for display.
 **
 *******************************************************************/

void FSCLKSetAlgorithmParameterText(int parameterNumber, unsigned char *theText)
{
    char *theCText;

    theCText = p2cstr(theText);
    switch (parameterNumber)
        {
        case kFSCLKAlgorithmEpsilon :
            ((FSCLKnetworkData *)theNet->data)->epsilon = atof(theCText);
            break;
        case kFSCLKAlgorithmFairness :
            ((FSCLKnetworkData *)theNet->data)->fairnessFactor = atof(theCText);
            break;
        }
}


/*******************************************************************
 ** FSCLKGetAlgorithmParameterText
 **
 ** Prints the algorithm specific variables to a string and returns that string for display.
 **
 *******************************************************************/

void FSCLKGetAlgorithmParameterText(int parameterNumber, unsigned char *theText)
{
    switch (parameterNumber)
        {
        case kFSCLKAlgorithmEpsilon :
            sprintf((char *)theText, "%f", ((FSCLKnetworkData *)theNet->data)->epsilon);
            break;
        case kFSCLKAlgorithmFairness :
            sprintf((char *)theText, "%f", ((FSCLKnetworkData *)theNet->data)->fairnessFactor);
            break;
        }
    c2pstr((char *)theText);
}


/*******************************************************************
 ** FSCLKResetAlgorithmParameters
 **
 ** Resets any network parameters so that learning operates in the same way it would
 ** have if the network was deleted and an identical network constructed.
 **
 *******************************************************************/
```

```
void FSCLKResetAlgorithmParameters()
{
    layer   *layerPtr;
    neuron *neuronPtr;

    layerPtr = theNet->layers;
    while (layerPtr != NULL)      /* Traverse layers */
        {
        neuronPtr = layerPtr->neurons;
        while (neuronPtr != NULL)  /* Traverse neurons */
            {
            ((FSCLKneuronData *)neuronPtr->data)->winningCount = 0;
            neuronPtr = neuronPtr->next;
            }
        layerPtr = layerPtr->next;
        }
}
```

A.4 Soft Competitive Learning Souce Code

A.4.1 SCL Header File (SCL.h)

```
/*******************************************************
 *                                                     *
 *  Claymore -- Soft Competitive Learning Algorithm    *
 *                                                     *
 *  This file contains the algorithm specific header information *
 *  for the SCL neural network algorithm.              *
 *                                                     *
 *******************************************************/

/* Algorithm Specific Resouce Numbers */
#define  rSCLAlgorithmSettingsDialog  134

enum {  /* Algorithm Settings Menu Item Numbers */
    kSCLAlgorithmEpsilon = 1,
    kSCLAlgorithmInitialVariance = 2,
    kSCLAlgorithmVarianceDecay = 3,
    kSCLAlgorithmMinimumVariance = 4
    };

struct SCLneuronData     /* NEURON */
  {
  float  totalInput;     /* Sum of all inputs to the neuron */
  float  expTotalInput;  /* Exponential of totalInput */
  float  variance;       /* Variance of the neurons RBF */
  };
typedef struct SCLneuronData  SCLneuronData;

struct SCLsynapseData     /* SYNAPSE */
  {
  float  difference;     /* Stores the value (weight-input) */
  };
typedef struct SCLsynapseData  SCLsynapseData;

struct SCLlayerData     /* LAYER */
  {
  int  numNeurons;    /* Number of neurons within this layer */
  float  normFactor;     /* Total of all exponential values */
  };
typedef struct SCLlayerData  SCLlayerData;

struct SCLnetworkData  /* NETWORK */
  {
  float epsilon;        /* Network learning rate. */
  float variance;
  float initialVariance;
  float varianceDecayFactor;
  float minimumVariance;
  };
typedef struct SCLnetworkData  SCLnetworkData;

#ifndef PI
#define  PI  3.141592654
#endif

/*** Prototypes ***/
int SCLinitNetworkData(network *theNetwork);
int SCLinitLayerData(layer *theLayer);
int SCLinitNeuronData(neuron *theNeuron);
int SCLinitSynapseData(synapse *theSynapse);
void SCLcomputeSums(network *net);
void SCLupdateActivations(network *net, dataset *theDataset);
void SCLupdateWeights(network *net);
void SCLdoEpoch(network *net, dataset *theDataset);
void SCLapplyVector(network *net, dataset *theDataset, int vectorNumber);
void SCLassembleStrings(network *net, char *theString);
void SCLSetAlgorithmParameterText(int parameterNumber, unsigned char *theText);
void SCLGetAlgorithmParameterText(int parameterNumber, unsigned char *theText);
void SCLResetAlgorithmParameters();
```

A.4.2 SCL Algorithm Code (SCL.c)

```
/************************************************************
 *                                                          *
 *  Claymore -- Soft Competitive Learning Algorithm         *
 *                                                          *
 *  This file contains the algorithm specific routines for the  *
 *  simulation of the SCL neural network algorithm.  It     *
 *  contains no code specific to the construction of the    *
 *  network it self. It only provides the functions necessary   *
 *  for learning.  Network construction routines reside in  *
 *  the sim.c module.(Adapted from Xerion source by Sue Becker)  *
 *                                                          *
 ************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "macSim.h"
#include "macSCL.h"
#include "macGlobals.h"

/******************************************************************
 ** SCLinitNetworkData
 **
 ** This function performs any initialization necessary when a
 ** network data record is created. (Algorithm dependent)
 **
 ******************************************************************/

int SCLinitNetworkData(network *theNetwork)
{
  if ((theNetwork->data = malloc(sizeof(SCLnetworkData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initNetworkData");
    return(MALLOC_FAILED);
    }
  ((SCLnetworkData *)theNetwork->data)->epsilon= 0.001;  /* Set an initial learning rate */
  ((SCLnetworkData *)theNetwork->data)->initialVariance = 0.0044;
  ((SCLnetworkData *)theNetwork->data)->variance = ((SCLnetworkData *)theNetwork->data)-
>initialVariance;
  ((SCLnetworkData *)theNetwork->data)->varianceDecayFactor = 0.0;
  ((SCLnetworkData *)theNetwork->data)->minimumVariance = 0.0044;
  return(0);
}


/******************************************************************
 ** SCLinitLayerData
 **
 ** This function performs any initialization necessary when a
 ** layer data record is created. (Algorithm dependent)
 **
 ******************************************************************/

int SCLinitLayerData(layer *theLayer)
{
  if ((theLayer->data = malloc(sizeof(SCLlayerData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initLayerData");
    return(MALLOC_FAILED);
    }
  ((SCLlayerData *)theLayer->data)->numNeurons = 0;
  return(0);
}


/******************************************************************
```

```
 ** SCLinitNeuronData
 **
 ** This function performs any initialization necessary when a
 ** neuron data record is created. (Algorithm dependent)
 **
 *****************************************************************/

int SCLinitNeuronData(neuron *theNeuron)
{
  if ((theNeuron->data = malloc(sizeof(SCLneuronData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initNeuronData");
    return(MALLOC_FAILED);
    }
  ((SCLneuronData *)theNeuron->data)->variance = ((SCLnetworkData *)theNet->data)->initialVariance;
  ((SCLneuronData *)theNeuron->data)->expTotalInput = 0.0;
  ((SCLneuronData *)theNeuron->data)->totalInput = 0.0;
  ((SCLlayerData *)theNeuron->layerPtr->data)->numNeurons++;
  return(0);
}


/*****************************************************************
 ** SCLinitSynapseData
 **
 ** This function performs any initialization necessary when a
 ** synapse data record is created. (Algorithm dependent)
 **
 *****************************************************************/

int SCLinitSynapseData(synapse *theSynapse)
{
  if ((theSynapse->data = malloc(sizeof(SCLsynapseData))) == NULL)
    {
    die("Unable to allocate necessary resources -- initSynapseData");
    return(MALLOC_FAILED);
    }
  return(0);
}


/*****************************************************************
 ** SCLcomputeSums
 **
 ** This function does the forward propagation through a network,
 ** computing the total input for each unit and determining the winner
 ** for each layer.  This is pass 1 of the network computations.  It
 ** returns the network error resulting from that operation.
 **
 *****************************************************************/

void SCLcomputeSums(network *net)
{
  layer   *layerPtr;
  neuron  *neuronPtr;
  synapse *synapsePtr;
  float   totalInput;

  layerPtr = net->layers;
  while (layerPtr != NULL)     /* Traverse the layers */
    {
    ((SCLlayerData *)layerPtr->data)->normFactor = 0.0;
    neuronPtr = layerPtr->neurons;
    while (neuronPtr != NULL) /* Traverse the neurons */
      {
      if (!(neuronPtr->lock))
        {
        totalInput = 0.0;
        synapsePtr = neuronPtr->synapseIn;
        while (synapsePtr != NULL)
          {
          ((SCLsynapseData *)synapsePtr->data)->difference =
                    synapsePtr->weight - synapsePtr->neuronIn->activation;
          totalInput += square(((SCLsynapseData *)synapsePtr->data)->difference);
          synapsePtr = synapsePtr->nextIn;
          }
        ((SCLneuronData *)neuronPtr->data)->totalInput = totalInput;
        ((SCLneuronData *)neuronPtr->data)->expTotalInput =
                        exp(- totalInput/(2.0*((SCLneuronData *)neuronPtr->data)->variance));
        ((SCLlayerData *)layerPtr->data)->normFactor += ((SCLneuronData *)neuronPtr->data)-
>expTotalInput;
        }
      neuronPtr = neuronPtr->next;
      }
    layerPtr = layerPtr->next;
    }
}


/*****************************************************************
 ** SCLupdateActivations
 **
 ** This function traverses the network and sets the activations of all
 ** neurons to the appropriate values.  It is called after computeSums.
 ** This is pass 2 of the network computations.
 **
 *****************************************************************/

void SCLupdateActivations(network *net, dataset *theDataset)
{
  layer   *layerPtr;
  neuron  *neuronPtr;

  layerPtr = net->layers->next;  /* Skip the first layer since it is the input layer */
  while (layerPtr != NULL)     /* Traverse layers */
    {
    neuronPtr = layerPtr->neurons;
    while (neuronPtr != NULL)  /* Traverse neurons */
      {
      neuronPtr->activation = ((SCLneuronData *)neuronPtr->data)->expTotalInput/
                              ((SCLlayerData *)layerPtr->data)->normFactor; /* Set activations */
      neuronPtr = neuronPtr->next;
      }
    net->error -= log( ((SCLlayerData *)layerPtr->data)->normFactor /
           (pow(sqrt(2.0 * PI * ((SCLnetworkData *)net->data)->variance), ((SCLlayerData *)layerPtr-
>data)->numNeurons) *
            ((SCLlayerData *)layerPtr->data)->numNeurons * theDataset->numPatterns));
    layerPtr = layerPtr->next;
    }
}


/*****************************************************************
 ** SCLupdateWeights
 **
 ** This function updates the weights for the winning unit in each layer.
 **
 *****************************************************************/

void SCLupdateWeights(network *net)
{
  layer   *layerPtr;
  neuron  *neuronPtr;
  synapse *synapsePtr;

  layerPtr = net->layers->next;
  while (layerPtr != NULL)    /* Traverse layers */
    {
    neuronPtr = layerPtr->neurons;
    while (neuronPtr != NULL)   /* Traverse neurons */
      {
      synapsePtr = neuronPtr->synapseIn;
      while (synapsePtr != NULL)  /* Traverse synapses */
        {
        synapsePtr->weight -= ((SCLnetworkData *)net->data)->epsilon * neuronPtr->activation *
                              ((SCLsynapseData *)synapsePtr->data)->difference;
        synapsePtr = synapsePtr->nextIn;
        }
      neuronPtr = neuronPtr->next;
      }
    layerPtr = layerPtr->next;
    }
}


/*****************************************************************
 ** SCLdoEpoch
 **
 ** Applies the set of input patterns to the network and calls SCLcomputeSums,
 ** SCLupdateActivations and SCLupdateWeights to perform the learning.
 **
 *****************************************************************/

void SCLdoEpoch(network *net, dataset *theDataset)
```

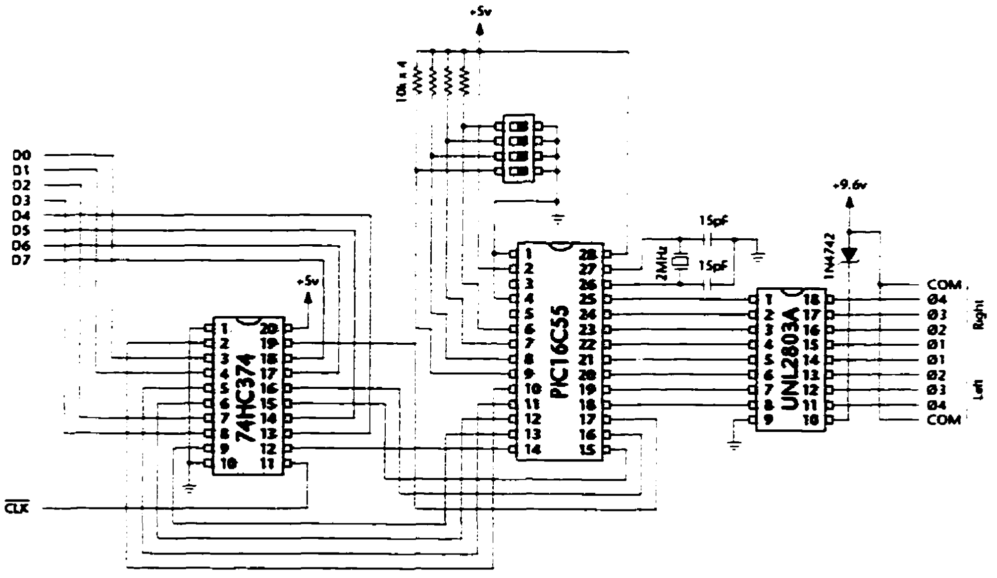
```
{
    int         index;
    layer  *theLayer;
    neuron      *theNeuron;
    dataElement *element;
    float   newVariance;

    /* Set pointer to the first input vector within the dataset. */
    element = theDataset->data;
    net->error = 0.0;           /* Clear the network error. */
    while(element != NULL)      /* Apply each input vector in turn. */
        {
        index = 0;
        theNeuron = net->layers->neurons;
        while(theNeuron != NULL)                              /* Set layer 0  */
            {                                                 /* activations  */
            theNeuron->activation = element->inputData[index++]; /* equal to the */
            theNeuron = theNeuron->next;                      /* the input    */
            }                                                 /* vector.      */
        element = element->next;

        SCLcomputeSums(net);         /* Computed the weighted sums for each neuron. */
        SCLupdateActivations(net, theDataset);   /* Update all neuron activations at once. */
        if (!theNet->batch)       /* If weights are to be updated in batch mode, */
            SCLupdateWeights(net);     /* then don't do it here. */
        }
    if (theNet->batch)          /* Update weights here when in batch mode. */
        SCLupdateWeights(net);
    /* Decay the variance if necessary */
    if (((SCLnetworkData *)theNet->data)->varianceDecayFactor != 0.0)
        {
        newVariance = ((SCLnetworkData *)theNet->data)->variance * ((SCLnetworkData *)theNet->data)-
>varianceDecayFactor;
        if (newVariance < ((SCLnetworkData *)theNet->data)->minimumVariance)
            newVariance = ((SCLnetworkData *)theNet->data)->minimumVariance;
        if (newVariance != ((SCLnetworkData *)theNet->data)->variance)
            {
            ((SCLnetworkData *)theNet->data)->variance = newVariance;
            theLayer = theNet->layers;
            while (theLayer != NULL)
                {
                theNeuron = theLayer->neurons;
                while (theNeuron != NULL)
                    {
                    ((SCLneuronData *)theNeuron->data)->variance = newVariance;
                    theNeuron = theNeuron->next;
                    }
                theLayer = theLayer->next;
                }
            }
        }
}


/*****************************************************************
 ** SCLapplyVector
 **
 ** Applies the specified vector number to the network and calls SCLcomputeSums and
 ** SCLupdateActivations to actually perform the computations.
 **
 *****************************************************************/

void SCLapplyVector(network *net, dataset *theDataset, int vectorNumber)
{
    int         index;
    neuron      *theNeuron;
    dataElement *element;

    /* Set pointer to the first input vector within the dataset. */
    element = theDataset->data;
    index = 0;
    while(index++ != vectorNumber)       /* Locate the specified  input vector. */
        element = element->next;
    theNeuron = net->layers->neurons;
    index = 0;
    while(theNeuron != NULL)                              /* Set layer 0  */
        {                                                 /* activations  */
        theNeuron->activation = element->inputData[index++]; /* equal to the */
        theNeuron = theNeuron->next;                      /* the input    */
        }                                                 /* vector.      */
    SCLcomputeSums(net);         /* Computed the weighted sums for each neuron. */
    SCLupdateActivations(net, theDataset);   /* Update all neuron activations at once. */
```

```
}

/*****************************************************************
 ** SCLassembleStrings
 **
 ** Prints the algorithm specific variables to a string and returns that string for display.
 **
 *****************************************************************/

void SCLassembleStrings(network *net, char *theString)
{
    if (net != NULL)
      sprintf(theString, "Soft Competitive Learning\r\rLearning Rate (epsilon) = %f\rInitial Variance
= %f\rVariance Decay Factor = %f\rMinimum Variance = %f\r\rVariance = %f\r",
            ((SCLnetworkData *)net->data)->epsilon, ((SCLnetworkData *)net->data)->initialVariance,
             ((SCLnetworkData *)net->data)->varianceDecayFactor,
             ((SCLnetworkData *)net->data)->minimumVariance,
             ((SCLnetworkData *)net->data)->variance);
}

/*****************************************************************
 ** SCLSetAlgorithmParameterText
 **
 ** Prints the algorithm specific variables to a string and returns that string for display.
 **
 *****************************************************************/

void SCLSetAlgorithmParameterText(int parameterNumber, unsigned char *theText)
{
    char *theCText;
    layer *theLayer;
    neuron *theNeuron;

    theCText = p2cstr(theText);
    switch (parameterNumber)
        {
        case kSCLAlgorithmEpsilon :
            ((SCLnetworkData *)theNet->data)->epsilon = atof(theCText);
            break;
        case kSCLAlgorithmInitialVariance :
            ((SCLnetworkData *)theNet->data)->initialVariance = atof(theCText);
            ((SCLnetworkData *)theNet->data)->variance = ((SCLnetworkData *)theNet->data)-
>initialVariance;
            theLayer = theNet->layers;
            while (theLayer != NULL)
                {
                theNeuron = theLayer->neurons;
                while (theNeuron != NULL)
                    {
                    ((SCLneuronData *)theNeuron->data)->variance = ((SCLnetworkData *)theNet->data)-
>initialVariance;
                    theNeuron = theNeuron->next;
                    }
                theLayer = theLayer->next;
                }
            break;
        case kSCLAlgorithmVarianceDecay :
            ((SCLnetworkData *)theNet->data)->varianceDecayFactor = atof(theCText);
            break;
        case kSCLAlgorithmMinimumVariance :
            ((SCLnetworkData *)theNet->data)->minimumVariance = atof(theCText);
            break;
        }
}

/*****************************************************************
 ** SCLGetAlgorithmParameterText
 **
 ** Prints the algorithm specific variables to a string and returns that string for display.
 **
 *****************************************************************/

void SCLGetAlgorithmParameterText(int parameterNumber, unsigned char *theText)
{
    switch (parameterNumber)
        {
        case kSCLAlgorithmEpsilon :
            sprintf((char *)theText, "%f", ((SCLnetworkData *)theNet->data)->epsilon);
            break;
        case kSCLAlgorithmInitialVariance :
            sprintf((char *)theText, "%f", ((SCLnetworkData *)theNet->data)->initialVariance);
            break;
        case kSCLAlgorithmVarianceDecay :
            sprintf((char *)theText, "%f", ((SCLnetworkData *)theNet->data)->varianceDecayFactor);
            break;
        case kSCLAlgorithmMinimumVariance :
            sprintf((char *)theText, "%f", ((SCLnetworkData *)theNet->data)->minimumVariance);
            break;
        }
    c2pstr((char *)theText);
}

/*****************************************************************
 ** SCLResetAlgorithmParameters
 **
 ** Resets any network parameters so that learning operates in the same way it would
 ** have if the network was deleted and an identical network constructed.
 **
 *****************************************************************/

void SCLResetAlgorithmParameters()
{
    layer *theLayer;
    neuron *theNeuron;

    ((SCLnetworkData *)theNet->data)->variance = ((SCLnetworkData *)theNet->data)->initialVariance;
    theLayer = theNet->layers;
    while (theLayer != NULL)
        {
        theNeuron = theLayer->neurons;
        while (theNeuron != NULL)
            {
            ((SCLneuronData *)theNeuron->data)->variance = ((SCLnetworkData *)theNet->data)-
>initialVariance;
            theNeuron = theNeuron->next;
            }
        theLayer = theLayer->next;
        }
}
```

Support Hardware and Source Code B

B.1 Stepper Motor Controller

B.1.1 Circuit Diagram



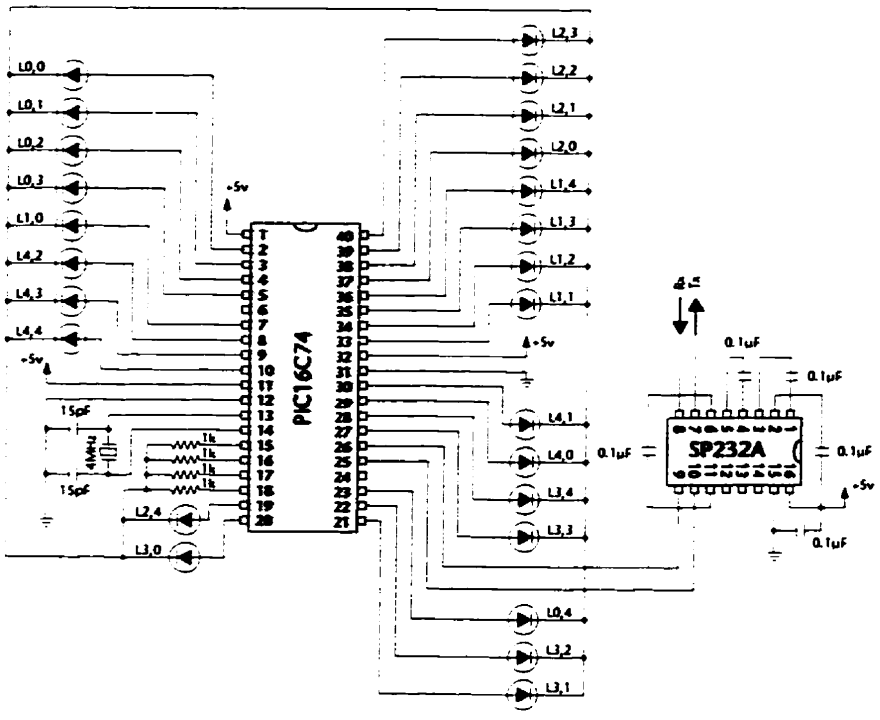
B.1.2 Stepper Controller Assembly Code

```
; STEPPER.ASM Modified: April 2, 1998

; Program to conrol a pair of stepper motors
; based on control information supplied by
; a HandyBoard

; Communication from the HandyBoard is received
; through PORTB.  HandyBoard writes an 8 bit value
; to a data latch on PORTB indicating the 3 bit
; speed and direction of each motor.

; Output to the motors is performed through PORTC
; which provides drive to the motors via a
; Darlington transistor arrangement.

        processor 16C55; set processor type

        constant porta = 0x5
        constant portb = 0x6
        constant portc = 0x7
        constant TMR0 = 0x1
        constant STATUS = 0x3

        constant speedreg = 0x1F
        constant speedl = 0x1E
        constant speedr = 0x1D
        constant countl = 0x1C
        constant countr = 0x1B
        constant drivel = 0x1A
        constant driver = 0x19
        constant tmrval = 0x18
        constant temp = 0x0F
        constant dirl = 0x3
        constant dirr = 0x7

START   ORG     0x00
        CLRW
        TRIS    portc           ; Configure PORTC as output
        MOVF    porta,0         ; Load the counter prescale value
        ANDLW   0x7             ; Discard unused bits
        OPTION                  ; Load the OPTION register
        MOVLW   0x03            ; Load W with left motor drive pattern
        MOVWF   drivel          ; Initialize left drive pattern
        MOVLW   0x30            ; Load W with right motor drive pattern
        MOVWF   driver          ; Initialize right drive pattern
        CLRF    speedreg        ; Initialize speed to zero
        CLRF    speedl
        CLRF    speedr

CHECK   MOVF    portb,0         ; Get the current speed from the HB
        SUBWF   speedreg,0      ; Test if speed is the same as before
        BTFSC   STATUS,2
        GOTO    DRIVE           ; If the same, goto drive routine
        MOVF    portb,0         ; Load the new speed value
        MOVWF   speedreg        ; Store the value in RAM
        ANDLW   0x7
        MOVWF   speedl          ; Save the left motor speed separately
        SWAPF   speedreg,0      ; Reload W with speed, swapping nibbles
        ANDLW   0x7
        MOVWF   speedr          ; Save the right motor speed separately
        COMF    speedreg,0      ; Load the complement of speed
        MOVWF   temp
        ANDLW   0x7
        MOVWF   countl          ; Initialize left countdown counter
        SWAPF   temp,0
        ANDLW   0x7
        MOVWF   countr          ; Initialize right countdown counter
        CLRF    temp

DRIVE   MOVF    temp,0          ; Load last temp stored timer value
        MOVWF   tmrval          ; Store it in timer location
        MOVF    TMR0,0          ; Get the current timer value
        MOVWF   temp            ; Save current timer value in temp location
        MOVF    tmrval,0        ; Load old timer value
        SUBWF   temp,0          ; Subtract old timer value from new value
        BTFSC   STATUS,0        ; If result negative timer has rolled over
        GOTO    DRIVE           ; If timer hasn't rolled, loop
```

```
CHECKL  MOVF    speedl.1         ; Load the speed value to update zero bit
        BTFSC   STATUS.2         ; Test if speed is zero
        GOTO    CHECKR           ; If zero go on to right motor control
        MOVF    countl.1         ; Load the counter to update zero bit
        BTFSC   STATUS.2         ; Test if counter is zero
        GOTO    LSTIM            ; If zero jump to left stimulus update
        DECF    countl.1         ; Decriment the left speed counter
        GOTO    CHECKR           ; Skip to the other motor routine
LSTIM   MOVF    drivel.0         ; Load the current drive pattern
        BTFSC   speedreg.dirl    ; Check if left is moving forward or back
        GOTO    LBACK            ; If not forward do backward
        RLF     drivel.1         ; Shift the drive stimulus left
        BCF     drivel.0         ; Zero the shifted in bit just in case
        BTFSC   drivel.4         ; Check if 1 shifted out of lower nibble
        BSF     drivel.0         ; If 1 shifted out. set low bit to 1
        GOTO    CLEANL
LBACK   RRF     drivel.1         ; Shift the drive stimulus right
        BTFSC   STATUS.0         ; Check if 1 shifted out of lower nibble
        BSF     drivel.3         ; If 1 shifted out. set high bit to 1
CLEANL  MOVLW   0x0F             ; Load W with mask 00001111
        ANDWF   drivel.1         ; Apply the mask to the left drive value
        COMF    speedl.0
        ANDLW   0x7
        MOVWF   countl           ; Reset the left counter

CHECKR  MOVF    speedr.1         ; Load the speed value to update zero bit
        BTFSC   STATUS.2         ; Test if speed is zero
        GOTO    STIMOUT          ; If zero. skip to output routine
        MOVF    countr.1         ; Load the counter to update zero bit
        BTFSC   STATUS.2         ; Test if counter is zero
        GOTO    RSTIM            ; If zero jump to right stimulus update
        DECF    countr.1         ; Decriment the right speed counter
        GOTO    STIMOUT
RSTIM   MOVF    driver.0
        BTFSS   speedreg.dirr    ; Check if right is moving forward or back
        GOTO    RBACK            ; If not forward do backward
        RLF     driver.1         ; Shift the drive stimulus left
        BTFSC   STATUS.0         ; Check if 1 shifted out of upper nibble
        BSF     driver.4         ; If 1 shifted out. set low bit to 1
        GOTO    CLEANR
RBACK   RRF     driver.1         ; Shift the drive stimuls right
        BCF     driver.7         ; Zero the shifted in bit just in case
        BTFSC   driver.3         ; Check if 1 shifted out of upper nibble
        BSF     driver.7         ; If 1 shifted out. set high bit to 1
CLEANR  MOVLW   0xF0             ; Load W with mask 11110000
        ANDWF   driver.1         ; Apply the mask to the right drive value
        COMF    speedr.0
        ANDLW   0x7
        MOVWF   countr           ; Reset the right counter

STIMOUT MOVF    drivel.0         ; Load the left stimulus value
        ADDWF   driver.0         ; Add the right stimuls value
        MOVWF   portc            ; Output the new drive stimulus
        GOTO    CHECK            ; Go back to the start

        end
```

B.2 Light Board Controller

B.2.1 Circuit Diagram



B.2.2 Light Board Assembly Code

```
; Light panel controller

; This program reads input from a serial
; line at 2400 baud and stores the received
; data in local memory.  Once the data has
; been read in, it is displayed on the light
; panel outputs in a cyclical fashion.

        processor 16C74; Set processor type.

; define some constants to make the code more
; readable.

        constant W = 0
        constant F = 1

        constant C = 0
        constant T0IF = 2
        constant BRGH = 2
        constant Z = 2
        constant SYNC = 4
        constant TXIE = 4
        constant TXIF = 4
        constant CREN = 4
        constant BANKB = 5
        constant RCIE = 5
        constant RCIF = 5
        constant TX89 = 6
        constant RC89 = 6
        constant SPEN = 7
        constant GIE = 7
        constant INDF = 0x00
        constant STATUS = 0x03
        constant FSR = 0x04
        constant PORTA = 0x05
        constant PORTB = 0x06
        constant PORTC = 0x07
        constant PORTD = 0x08
        constant PORTE = 0x09
        constant INTCON = 0x0B
        constant PIR1 = 0x0C
        constant T1CON = 0x10
        constant OPTREG = 0x81
        constant TRISA = 0x85
        constant TRISB = 0x86
        constant TRISC = 0x87
        constant TRISD = 0x88
        constant TRISE = 0x89
        constant RCSTATUS = 0x18
        constant TXREG = 0x19
        constant RCREG = 0x1A
        constant ADCON0 = 0x1F
        constant PIE1 = 0x8C
        constant TXSTATUS = 0x98
        constant SPBRG = 0x99
        constant ADCON1 = 0x9F

        constant W_TEMP = 0x20          ; May also be A0. depending on STATUS
                                        ; when interrupt is received.
        constant STATUS_TEMP = 0x21
        constant NUMPATS = 0x22         ; Number of patterns stored.
        constant PATCOUNT = 0x23        ; Current pattern being displayed.
        constant TEMP1 = 0x24           ; Temporary storage.
        constant TEMP2 = 0x25           ; Temporary storage.
        constant SPEED = 0x26
        constant SCOUNT = 0x27
        constant FSRTEMP = 0xA1
        constant RCPTR= 0xA2            ; Ptr to next addr for pattern storage
        constant ROW1TEMP = 0xA3
        constant ROW2TEMP = 0xA4
        constant ROW3TEMP = 0xA5
        constant ROW4TEMP = 0xA6
        constant ROW5TEMP = 0xA7

        ; ***** End of Constant Definitions *****


        ORG     0x00
        GOTO    START

        ORG     0x04
        GOTO    IHNDLR          ; Install interrupt handler vector.

START   CLRF    INTCON          . Initially disable all interrupts.

                                ; ** INITIALIZE PORT DIRECTIONS **
        BSF     STATUS,BANKB    ; Select Bank 1.
        CLRF    TRISA           ; Configure PORTA as outputs.
        CLRF    TRISB           ; Configure PORTB as outputs.
        CLRF    TRISD           ; Configure PORTD as outputs.
        CLRF    TRISE           ; Configure PORTE as outputs.
        MOVLW   B'10000000';
        MOVWF   TRISC           ; Configure PORTC as outputs (except TX & RC).
        MOVLW   B'00000110'     ; Load Analog Port configuration bit pattern.
        MOVWF   ADCON1          ; Set all A/D ports to digital I/O

        MOVLW   B'00000111'     ; Load TMR0 configuration value.
        MOVWF   OPTREG          ; Store configuration bits for TMR0.

                                ; ** INITIALIZE SERIAL PORT **
        MOVLW   0x19            ; Load value for Baud Rate Generator.
        MOVWF   SPBRG           ; Set Baud Rate Generator at 2400 baud.
        CLRF    TXSTATUS        ; Clear high baud rate bit and enable async mode.
        BCF     STATUS,BANKB    ; Select Bank 0.
        MOVLW   B'10010000'     ; Enable serial port for receipt of 8 bit words.
        MOVWF   RCSTATUS;
        BSF     STATUS,BANKB    ; Select Bank 1.
;       BSF     PIE1,TXIE       ; Enable transmit interrupts.
```

```
          CLRF     PIE1           ; Clear all interrupt enable bits.
          BSF      PIE1.RCIE      ; Enable receive interrupts.

          MOVLW    0x28           ; ** INITIALIZE PATTERN POINTER AND COUNTERS **
          MOVWF    FSRTEMP        ; Point the next pattern pointer to start of table.
          MOVWF    RCPTR          ; Point new pattern storage pointer to start of table.
          CLRF     ROW1TEMP       ; Clear the new pattern temporary storage location.
          CLRF     ROW2TEMP       ;
          CLRF     ROW3TEMP       ;
          CLRF     ROW4TEMP       ;
          CLRF     ROW5TEMP       ;
          BCF      STATUS.BANKB   ; Select Bank 0.
          CLRF     NUMPATS        ; Zero number of patterns value.
          CLRF     PATCOUNT       ; Zero pattern count.
          CLRF     SPEED          ; Zero speed counter target.
          CLRF     PORTA          ; Clear the outputs.
          CLRF     PORTB          ;
          CLRF     PORTC          ;
          CLRF     PORTD          ;
          CLRF     PORTE          ;
          CLRF     ADCON0         ; Turn off A/D converter, since we don't need it.

          MOVLW    B'11000000'    ;
          MOVWF    INTCON         ; Enable global and peripheral interrupts.

DISPLAY   BCF      INTCON.GIE     ; Disable Global Interrupt flag.
          BTFSC    INTCON.GIE     ; Ensure flag was cleared.
          GOTO     DISPLAY        ; If not cleared, try again.
          MOVF     NUMPATS.F      ; Touch pattern count to permit zero test.
          BTFSC    STATUS.Z       ; Test if pattern count not zero.
          GOTO     ENABLEI        ; If zero skip to end of display routine.
          BSF      STATUS.BANKB   ; Select Bank 1.
          MOVF     FSRTEMP.W      ; Load the stored FSR value.
          MOVWF    FSR            ; Store FSR for indirect addressing.
          BTFSS    FSRTEMP.7      ; Check if FSR in Bank 0 or Bank 1.
          GOTO     SEL0           ; If FSR Bank 0 skip to Bank 0 select.
          BSF      STATUS.BANKB   ; Select Bank 1.
          GOTO     DISPST         ; Begin displaying the pattern.
SEL0      BCF      STATUS.BANKB   ; Select Bank 0.
DISPST    MOVF     INDF.W         ; Read memory pointed to by FSR.
          MOVWF    PORTA          ; Store first line in PORTA.
          BTFSS    INDF.4         ; Test if bit 4 is set.
          GOTO     FIX
          BSF      PORTC.4
          GOTO     LINE2
FIX       BCF      PORTC.4
LINE2     INCF     FSR.F
          MOVF     INDF.W         ; Load next line.
          MOVWF    PORTB          ; Store second/third line in PORTB.
          INCF     FSR.F
          MOVF     INDF.W         ; Load next line.
          MOVWF    PORTD          ; Store third/fourth line in PORTD.
          INCF     FSR.F
          MOVF     INDF.W         ; Load last line fragment.
          MOVWF    PORTE          ; Store last part of line four.
          INCF     FSR.F
          BTFSC    STATUS.BANKB   ; See if we are in Bank 0.
          GOTO     TESTEND
          BTFSS    FSR.7          ; Test if FSR has hit the end of Bank 0.
          GOTO     TESTEND
          MOVLW    0xA8           ; Load base pattern address for Bank 1.
          MOVWF    FSR            ; Store new FSR value.
TESTEND   BCF      STATUS.BANKB   ; Select Bank 0.
          INCF     PATCOUNT.F     ; Incriment the current pattern number.
          MOVF     PATCOUNT.W     ; Get current pattern number.
          SUBWF    NUMPATS.W      ; Subtract current pat num from total patterns.
          BTFSS    STATUS.Z       ; Test if current pattern = last pattern.
          GOTO     SAVEFSR        ; If not at last pattern, save current FSR.
          CLRF     PATCOUNT       ; Reset pattern counter to zero.
          MOVLW    0x28           ; Load first pattern address into W.
          MOVWF    FSR            ; Store new value in FSR register.
SAVEFSR   MOVF     FSR.W          ; Get the current value of FSR.
          BSF      STATUS.BANKB   ; Select Bank 1.
          MOVWF    FSRTEMP        ; Save it for later use.
ENABLEI   BSF      STATUS.BANKB   ; Select Bank 1.
          BSF      INTCON.GIE     ; Re-enable Global Interrupt flag.
WAIT      MOVF     RCPTR.F        ; Touch receive pointer to update flags.
          BTFSS    STATUS.Z       ; Test if receive pointer zero
          CALL     CHK4PAT        ; If !zero, table has room. Check for new pattern.
          BTFSS    INTCON.T0IF    ; Test if TMR0 has overflown.
          GOTO     WAIT           ; If not overflown, wait.
          BCF      INTCON.T0IF    ; Clear overflow flag for TMR1.
```

```
         BCF     STATUS,BANKB     ; Select Bank 0.
         INCF    SCOUNT,F         ; Increment speed counter.
         MOVF    SPEED,W          ; Load W with speed value.
         SUBWF   SCOUNT,W         ; Subtract target count from current count.
         BTFSC   STATUS,C         ; If carry clear, count is less than target.
         GOTO    CLRCNT           ; If we've reached the target, clear counter and display.
         BSF     STATUS,BANKB     ; Select Bank 1.
         GOTO    WAIT             ; Wait until count equals target.
CLRCNT   CLRF    SCOUNT           ; Zero the counter.
         GOTO    DISPLAY          ; Update display.

                                  ; *** Check for a new pattern and insert it  ***
                                  ; *** into the pattern table.                ***
                                  ; *** We should be in Bank 1 upon entering   ***
                                  ; *** this routine.                          ***
CHK4PAT  MOVLW   ROW1TEMP         ; Get base address of temp. pattern buffer.
         MOVWF   FSR              ; Prepare for indirect addressing.
CHKLOOP  MOVF    INDF,F           ; Touch memory location to permit zero test.
         BTFSC   STATUS,Z         ; Test if location was zero (empty).
         RETURN                   ; If zero we don't have a complete pattern, so return.
         MOVLW   ROW5TEMP         ; Load W with top address of temp array.
         SUBWF   FSR,W            ; Subtract end pointer value from present value.
         BTFSC   STATUS,C         ; if Carry set they're equal, so we have full pattern.
         GOTO    DISABLI          ; We must now condense the five row bytes into four.
         INCF    FSR,F            ; Increment memory pointer.
         GOTO    CHKLOOP          ; If more to test, test them.
DISABLI  BCF     INTCON,GIE       ; Disable Global Interrupt flag.
         BTFSC   INTCON,GIE       ; Ensure flag was cleared.
         GOTO    DISABLI          ; If not cleared, try again.
SHUFFLE  MOVLW   B'00011111'      ; Load mask.
         ANDWF   ROW1TEMP,F       ; Clear the three MSBs in row 1.
         ANDWF   ROW2TEMP,F       ; Clear the three MSBs in row 2.
         ANDWF   ROW4TEMP,F       ; Clear the three MSBs in row 4.
         BTFSC   ROW2TEMP,0       ; Test if LSB of row 2 is set.
         BSF     ROW1TEMP,5       ; If set, set bit 5 in port 1 pattern to match.
         BCF     STATUS,C         ; Clear the carry bit.
         RRF     ROW2TEMP,F       ; Rotate row 2 pattern right 1 bit.
         SWAPF   ROW3TEMP,W       ; Swap row 3 pattern value into W.
         ANDLW   B'11110000'      ; Mask upper nibble of W.
         ADDWF   ROW2TEMP,F       ; Add row 3 bits to port 2 pattern.
         SWAPF   ROW3TEMP,F       ; Swap high and low nibbles of row 3.
         MOVLW   B'00000001'      ; Load mask.
         ANDWF   ROW3TEMP,F       ; Discard all but LSB of port 3 pattern.
         BCF     STATUS,C         ; Clear the carry bit.
         RLF     ROW4TEMP,W       ; Rotate row 4 left 1 bit into W.
         ADDWF   ROW3TEMP,F       ; Add row 4 bits to port 3 pattern.
         SWAPF   ROW5TEMP,W       ; Swap row 5 into W.
         MOVWF   ROW4TEMP         ; Store temporarily in port 4 pattern.
         RLF     ROW4TEMP,F       ; Rotate left 1 bit.
         RLF     ROW4TEMP,F       ; Rotate left 1 bit.
         MOVLW   B'11000000'      ; Load mask.
         ANDWF   ROW4TEMP,W       ; Apply mask, load result into W.
         ADDWF   ROW3TEMP,F       ; Add row 5 bits to port 3 pattern.
         RRF     ROW5TEMP,F       ; Rotate row 5 right 1 bit.
         RRF     ROW5TEMP,F       ; Rotate row 5 right 1 bit.
         MOVLW   B'00000111'      ; Load mask.
         ANDWF   ROW5TEMP,W       ; Apply mask and place result in W.
         MOVWF   ROW4TEMP         ; Store result in port 4 pattern.
         MOVLW   ROW1TEMP         ; Load start address of pattern array.
         MOVWF   ROW5TEMP         ; Store address in temporary location.
XFERPAT  MOVF    ROW5TEMP,W       ; Get pointer.
         MOVWF   FSR              ; Prepare for indirect addressing.
         BTFSC   RCPTR,7          ; Test if address for new pattern is Bank 1.
         GOTO    STOREB1          ; If in Bank 1 go to Bank 1 storage routine.
STOREB0  MOVF    INDF,W           ; Load port pattern into W.
         CLRF    INDF             ; Clear the row pattern location.
         BCF     STATUS,BANKB     ; Switch to Bank 0.
         MOVWF   TEMP1            ; Store port pattern in temporary location.
         BSF     STATUS,BANKB     ; Switch back to Bank 1.
         MOVF    RCPTR,W          ; Load the pattern table storage location.
         MOVWF   FSR              ; Prepare for indirect addressing.
         BCF     STATUS,BANKB     ; Switch back to Bank 0.
         MOVF    TEMP1,W          ; Reload port pattern from temporarly location.
         MOVWF   INDF             ; Store port pattern in final location.
         BSF     STATUS,BANKB     ; Switch to Bank 1.
         GOTO    STORED           ;
STOREB1  MOVF    INDF,W           ; Load port pattern into W.
         CLRF    INDF             ; Clear the row pattern location.
         MOVWF   W_TEMP           ; Store port pattern in temporary location.
         MOVF    RCPTR,W          ; Load the pattern table storage location.
         MOVWF   FSR              ; Prepare for indirect addressing.
         MOVF    W_TEMP,W         ; Reload port pattern from temporary location.
```

```
        MOVWF   INDF            ; Store port pattern in final location.
STORED  INCF    RCPTR,F         ; Increment table storage location pointer.
        INCF    ROW5TEMP,F      ; Increment temporary port pattern pointer.
        MOVLW   ROW5TEMP        ; Load end address of pattern array + 1.
        SUBWF   ROW5TEMP,W      ; Subtract current address from end address.
        BTFSS   STATUS,C        ; If carry set, end was reached so copy is done.
        GOTO    XFERPAT         ; If not done transfer next port pattern.
        CLRF    ROW5TEMP        ; Clear row 5 pattern location.
        BCF     STATUS,BANKB    ; Select Bank 0.
        INCF    NUMPATS,F       ; Increment number of stored patterns.
        BSF     INTCON,GIE      ; Re-enable global interrupts.
        BSF     STATUS,BANKB    ; Select Bank 1.
        MOVLW   0x80            ; Load Bank 0 rollover value.
        SUBWF   RCPTR,W         ; Subtract it from table storage location pointer.
        BTFSS   STATUS,Z        ; If not zero we haven't rolled over.
        RETURN
        MOVLW   0xA8            ; Load Bank 1 base address.
        MOVWF   RCPTR           ; Store value for proper storage of next pattern.
        RETURN


                                ; ** INTERRUPT HANDLER **
IHNDLR  MOVWF   W_TEMP          ; Save W value (May be Bank 0 or 1).
        SWAPF   STATUS,W        ; Swap STATUS into W to avoid modification.
        BCF     STATUS,BANKB    ; Switch to BANK 0.
        MOVWF   STATUS_TEMP     ; Save (swapped) STATUS value.
        BTFSC   PIR1,TXIF       ; Check if interrupt due to empty TX Reg.
        CALL    DO_TX           ; Branch to transmit routine.
        BTFSC   PIR1,RCIF       ; Check if interrupt due to data in RC Reg.
        CALL    DO_RC           ; Branch to receive routine.
        BCF     STATUS,BANKB    ; Ensure we are in Bank 0.
        SWAPF   STATUS_TEMP,W   ; Swap original STATUS into W.
        MOVWF   STATUS          ; Restore original STATUS register value.
        SWAPF   W_TEMP,F        ; Swap W_TEMP (no modification to STATUS).
        SWAPF   W_TEMP,W        ; Swap original W value into W register.
        RETFIE                  ; Return from interrupt.

DC_TEST BCF     STATUS,BANKB    ; Select Bank 0.
        MOVF    RCREG,W
        MOVWF   0x2C
        MOVF    PIR1,W
        MOVWF   TEMP1
        RLF     TEMP1,W
        MOVWF   0x2E
        BCF     PIR1,RCIF
        RETURN

DO_TX   NOP
        RETURN

DO_RC   BCF     STATUS,BANKB    ; Select Bank 0.
        BCF     PIR1,RCIF       ; Clear the interrupt flag.
        MOVF    RCREG,W         ; Get the received word.
        MOVWF   TEMP1           ; Store word in temp. loc. to permit tests.
        ANDLW   B'11100000'     ; Discard all but three MSBs.
        MOVWF   TEMP2           ; Store the three MSBs.
        SWAPF   TEMP2,F         ; Swap low and high nibble.
        BCF     STATUS,C        ; Clear the Carry Bit.
        RRF     TEMP2,F         ; Rotate the temp register right 1 bit.
        MOVLW   0x05
        SUBWF   TEMP2,W         ; Subtract 5 from three MSBs of received word.
        BTFSS   STATUS,C        ; If Carry is set, we have a control word,
        GOTO    STOREIT         ; otherwise MSBs are data index values.
        BTFSS   STATUS,Z        ; If Zero bit was set, control is speed.
        GOTO    TESTCLR         ; If not speed command, clear command?
        MOVF    TEMP1,W         ; Load received word.
        ANDLW   B'00011111'     ; Discard control bits.
        MOVWF   SPEED           ; Save speed value.
        RETURN
TESTCLR MOVLW   B'00000111'     ; Load mask.
        XORWF   TEMP2,W         ; And mask and command bits.
        BTFSS   STATUS,Z        ; If result not zero, intensity command.
        GOTO    INTENSE         ;
        CLRF    NUMPATS         ; Set number of patterns to zero.
        CLRF    PATCOUNT        ; Set next pattern to display to zero.
        MOVLW   0x28            ; Load table base address.
        BSF     STATUS,BANKB    ; Switch to Bank 1.
        MOVWF   RCPTR           ; Set new pattern storage pointer to base address.
        MOVWF   FSRTEMP         ; Set pattern display pointer to base address.
        CLRF    ROW1TEMP        ; Clear any partially received pattern.
        CLRF    ROW2TEMP        ;
        CLRF    ROW3TEMP        ;
        CLRF    ROW4TEMP        ;
        CLRF    ROW5TEMP        ;
        BCF     STATUS,BANKB    ; Switch to Bank 0.
        CLRF    PORTA           ; Clear the output ports to clear board.
        CLRF    PORTB           ;
        BCF     PORTC,4         ; Can't simply clear Port C because it controls RC too.
        CLRF    PORTD           ;
        CLRF    PORTE           ;
        RETURN
INTENSE MOVLW   B'00001111'     ; Load mask.
        ANDWF   TEMP1,F         ; Discard top nibble.
        BSF     TEMP1,7         ; Set bit for serial RX pin (RC7).
        MOVF    TEMP1,W         ; Load intensity pattern into W.
        BSF     STATUS,BANKB    ; Select Bank 1.
        MOVWF   TRISC           ; Store new intensity (via PORTC).
        RETURN
STOREIT MOVLW   ROW1TEMP        ; Load W with temporary storage base address.
        ADDWF   TEMP2,W         ; Add the pattern number to base address.
        MOVWF   FSR             ; Prepare of indirect addressing.
        MOVF    TEMP1,W         ; Load original received value.
        ANDLW   B'00011111'     ; Strip three MSBs (control/index bits).
        ADDLW   B'11100000'     ; Set the three MSBs of the _attern.
        BSF     STATUS,BANKB    ; Switch to Bank 1.
        MOVWF   INDF            ; Store the value.
        RETURN

        end
```

IMAGE EVALUATION
TEST TARGET (QA-3)