Chapter 5

Specification

5.1 Introduction

In this chapter, extensions to the initial environment that are required to make perception and action possible are described. The overall structure of the controller architecture is then specified. This is followed by a description of the training strategy to be used in enabling the robot to perform the desired target behaviour. Finally, the Matlab implementation of the learning system is described in detail. Although the BAT methodology does not stipulate this last item as a part of the Specification stage, it was deemed appropriate to discuss it in detail in this chapter.

5.2 Modifications to the Environment

The sensorimotor apparatus of the Khepera base unit is fixed. The light and proximity sensors are based on the Siemen's SFH900 miniature light reflection switch. This unit transmits and receives infra-red light (used for proximity measurement) and can also measure the amount of ambient light present. However, the transmitted infra-red is not modulated so the unit is susceptible to picking up stray infra-red from other light sources and these readings can be interpreted as obstacles. This situation precludes the use of the device where it is required to approach a light while avoiding obstacles (ie, the current direction of this work) since normal tungsten filament a.c. or d.c. lamps radiate large amounts of infra-red light. The light source will appear as an obstacle.

The target behaviour was made possible by the fitting of optical filters in front of d.c. powered lamps. A number of heat absorbing filters and hot mirrors were tested in various light intensity situations and several removed enough infra-red to allow the robot to approach the light bulb without seeing it as an object. Besides lining the walls of the playpen with Lego blocks (the aluminum coating was found to be too reflective), no other modifications to the environment were required.

5.3 The Controller Architecture

The controller architecture chosen to implement the desired behaviour should parallel the organization of the structured behaviour as determined in the *Behaviour Analysis* stage. Thus, a behaviour module (BM) was allocated to each of the structured behaviours *seeklight* and *avoidobst*. These two BM's were connected in a *flat* architecture with *integrated* outputs as shown in Figure 5-1 below. The integration function is determined by the suppressive relationship between *seeklight* and *avoidobst* as indicated in equation 4-1. The integration takes the form of a simple subsumption or switch which gives priority to obstacle avoidance when required.

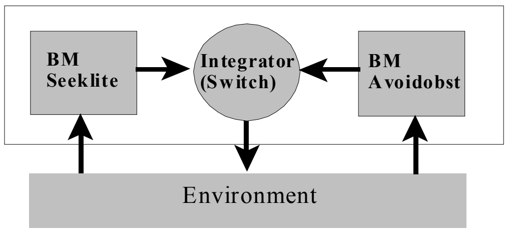

Figure 5-1
Flat Architecture with Integrated Outputs

5.4 The Training Strategy

The training strategy is specified by the *shaping policy* and *reinforcement program*. The shaping policy deals with whether the structured behaviour should be learned in a one-shot learning, or by a sequence of learning sessions in which each of the individual behaviours are trained independently. The reinforcement program determines what information will be provided to the learning system to make the robot converge to the target behaviour. These two aspects of the training strategy for this work are covered in the following sections.

5.4.1 Shaping Policy

In this work, *modular shaping* (as defined in the BAT methodology) was used to train the controller. That is, each behaviour module was trained individually in its respective task and then "frozen" (ie, training mode is turned off). The trained modules were then integrated into the control architecture of Figure 5-1.

5.4.2 Reinforcement Program

The BAT methodology assumes that the learning system used for each behaviour module is based on reinforcement learning. Specifically, it assumes that the architecture learns by *immediate reinforcement*. In this setting, there are three elements: the *learning system*, the *trainer*, and the *environment*. The job of the trainer is to make the actions chosen by the learning system in its environment converge to a predefined target behaviour.

In principle, the trainer could be a human observing the agent's interaction with the environment and issuing the required reinforcements to shape the behaviour. However, this scenario would be slow since the robot would have to wait at every iteration for the human to evaluate its last action and provide the necessary reinforcement signal. Instead, reinforcements are provided automatically by a *reinforcement program* or RP which, in this work, is a separate Matlab function of its own. An RP must have its own sensors or at least access to the same sensors as the learning system it is shaping. It's sensors play the critical role of allowing it to observe and evaluate the behaviour of the learning system. Using the sensor input, an RP evaluates the observed behaviour, generates a reinforcement value, and passes this value to the learning system for use in adapting its behaviour. This relationship is shown in figure 5-2. In this work, RP and *trainer* are used interchangeably. There is one RP per behaviour module and each RP makes use of the same eight light/proximity sensors as does the behaviour module it is shaping since no other sensing capabilities ship with the Khepera base unit.

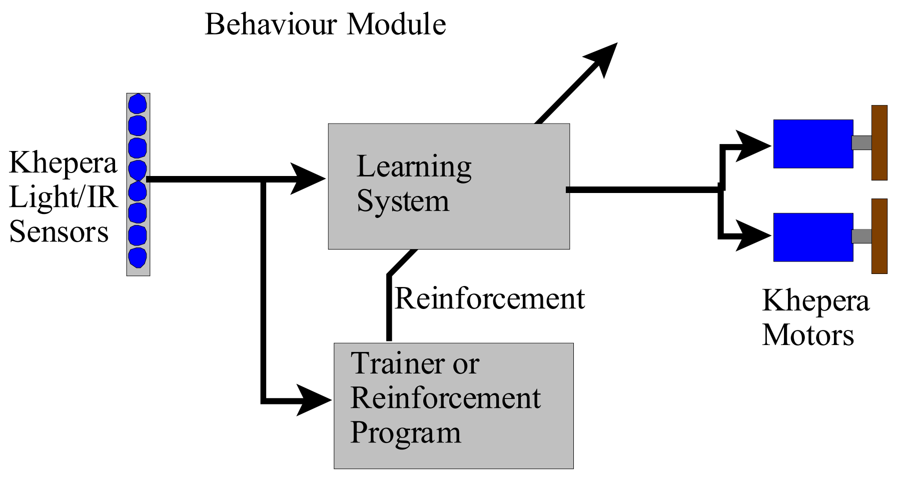

Figure 5-2
The Learning System-Trainer Relationship

5.5 Learning System Design and Implementation

5.5.1 Introduction

A section detailing the design and implementation of the learning system forming the heart of each behaviour module is not specifically prescribed in the BAT methodology. Instead, the methodology assumes that the learning system is already chosen and well described. However, in any engineering treatise based on the BAT methodology, implementation details of the learning system must be provided. This last section of the specification stage was created to provide these details.

In Chapter 2, it was established that recurrent neural networks were going to be used as learning system for each behaviour module and that these networks would be trained using Complementary Reinforcement Back-Propagation (CRBP). The next section provides implementation details for the `seeklight` and `avoidobst` behaviours. This is followed by a section providing implementation details on how these behaviours are integrated into the overall controller architecture of Figure 5-1.

5.5.2 Behaviour Module Design and Implementation

5.5.2.1 Introduction

This section describes the implementation of the `avoidobst` and `seeklite` behaviour modules in the Matlab programming environment. This section begins with a block diagram overview of the implementation of a behaviour module. Both behaviour modules share the structure and the functions described in this block diagram. The only difference between individual behaviour modules is the specification of the *Reinforcement Program* and the sensor input representation required by the neural network in each behaviour module. This section continues on to describe each individual behaviour module in terms of these differences.

5.5.2.2 Module Block Diagram Overview

A block diagram of a typical behaviour module is shown in Figure 5-3. The module is broken down into its two main components: the *learning system* and the *trainer*. Each block within the trainer and the learning system is labelled with a description of its function and the name of the Matlab m-file (in bold font) which performs that function. Listings of the m-files can be found in Attachment 1, *Matlab Functions for Obstacle Avoidance/Light Seeking*. Arrows indicate the flow of the algorithm and of information. The m-file which organizes the functions as shown in Figure 5-3 is named to reflect the purpose of the module (ie **seeklite.m** and **avoidobj.m**) and are included in Attachment 1 as well.

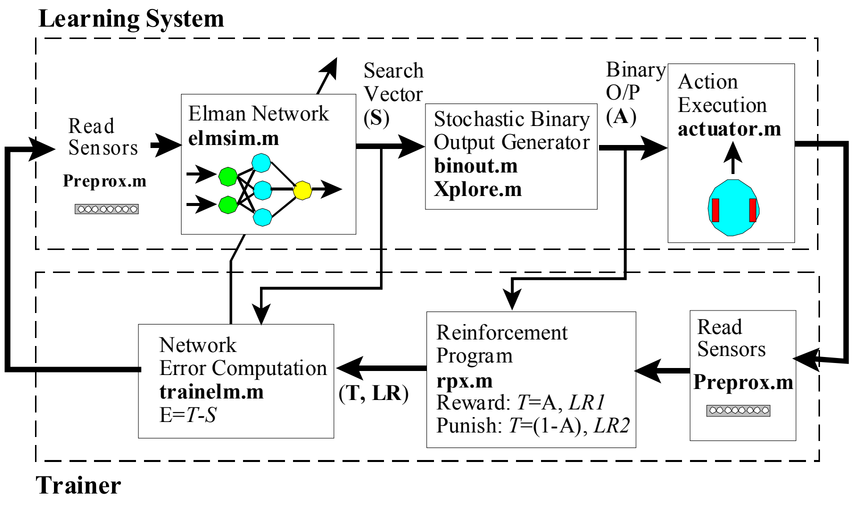

Figure 5-3
Block Diagram of Typical Behaviour Module

Beginning at the top left of Figure 5-3, the data required by the learning system is read from the sensor array of the Khepera. This data is passed through an *input representation* function which filters it as required for use by the next functional block. The function name shown is **prepro*x*.m**. Since each behaviour module may require a different input representation, the character *x* in the m-file name is changed to individualize the function. Of course, if the inputs are used without modification, this m-file is simply the identity function.

The input data is passed to the Elman network which is implemented by the m-file **elmsim.m**. This function computes the Elman network response, *S*, to the input vector supplied by **prepro*x*.m**. The structure of the Elman network and its interconnection to the sensors and actuators is shown in Figure 5-4 below which shows a detailed connectivity and data flow diagram of the Learning System portion of Figure 5-3.

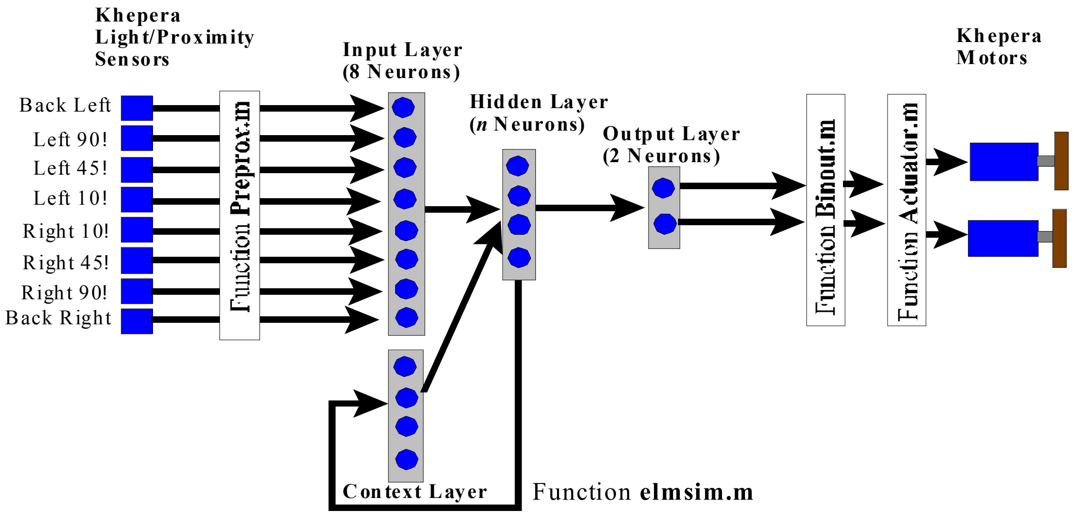

Figure 5-4
Elman Network Connectivity

The function **elmsim.m** was originally the Matlab m-file **simuelm.m** from the *Neural Networks Toolbox*. However, several modifications were required to make it work with this implementation. First, **simuelm.m** took as inputs a *matrix* where each column of the matrix represented one data vector for which a network response had to be computed. This was inappropriate since we wished to pass one vector of data to the network at a time and have it compute its response on that vector only. Second, it took the initial value of the context layer as an input argument. If no value was specified, then the context layer was set to zero. This meant that the current value of the context layer would have to be tracked outside of the function or it would otherwise be initialized to zero on every iteration. This was changed so that the function would initialize and keep track of the context layer on its own.

Another required change was to modify the default transfer functions of the hidden and output layers of the Elman network supplied with the *Neural Network Toolbox*. The default transfer function of the output layer of **simuelm.m** was the **purelin.m** (linear) function. This produces real valued outputs on (-! ,!). However, it was desired to constrain the outputs of the neworks in this thesis to (0 1). The output layer transfer function was therefore changed to the *Neural Network Toolbox* **logsig.m** (sigmoid) function.

The transfer function of the hidden layer was the **tansig.m** (hyperbolic tangent sigmoid) function. Although this was the proper activation function for this layer, it still required a small modification. The argument *tscale* was added to the **tansig.m** function to allow global control over the scaling of the sigmoid function. This change was incorporated to prevent a phenomenon known as *premature saturation* from occuring in the neurons of that layer. Premature saturation, and the role of *tscale* in its prevention, is more rigorously discussed in Chapter 6.

The response, or search vector *S*, of the Elman network (a two element vector) is passed to either **binout.m** or **xplore.m** (depending on the type of exploration function desired). The function **binout.m** uses a uniform random number generator to stochastically produce the binary valued action vector *A*. The function **xplore.m** uses a Gaussian random number generator to produce the vector *A*. The two element binary vector *A* is passed to **actuator.m** which causes one of four possible types of motion on the Khepera as shown in Table 5-1.

Binary Vector *O*	Khepera Action
[1 1]	move forward
[0 0]	move backward
[1 0]	turn right
[0 1]	turn left

Table 5-1
Binary Vector Action Selection

The m-file **actuator.m** not only executes the action on the Khepera but it also determines the granularity of each Khepera motion through the variable *mot_delay*. This variable controls the length of a loop delay after each action is executed.

In the *trainer* block, **preprox.m** is called to sample the Khepera sensors. The function

rp*x*.m (where *x* is used to distinguish among different reinforcement programs) uses the sampled sensor data to evaluate the action just performed by the learning system. This function is the "heart and soul" of the trainer and the code within it must be constructed in a fashion that causes the Elman network to converge to the target behaviour. It uses the action vector *A* from the learning system to compute the target vector and learning rate that will be needed to train the network in accordance with the Complementary Reinforcement Back-Propagation algorithm. It determines the target vector, *T*, and learning rate, *LR*, as shown in Table 5-2.

The action just performed was:	Target vector, learning rate:
Rewarded	*T*=*A*, Learning Rate = LR1
Punished	*T*=1-*A*, Learnining Rate = LR2
Neither rewarded or punished	*T*=*A*, Learning Rate = 1

Table 5-2
Determination of T and LR in **rpx.m**

LR1 and *LR2* are the learning rate for reward and punishment respectively. The target vector, *T,* and the learning rate, *LR*, are passed to **trnelm1p.m** to train the Elman network.

The function **trnelm1p.m** was originally the Matlab m-file **trainelm.m**. However, like **elmsim.m**, the original function treated the training task as a batch operation. That is, all the training data was supplied to **trainelm.m** in the form of columns of an input matrix and the function worked to train the network to the desired accuracy. The function **trnelm1p.m** accepts a target vector, *T*, and learning rate, *LR*, from **rp*x*.m** and provides a single error back-propagation to the network. Note that if **rp*x*.m** passes a learning rate value of 1, **trnelm1p.m** interprets this as a condition of no training and the network parameters are left unchanged.

After the network parameters are modified, the cycle begins again with a sampling of the sensors by **preprox.m**. in the learning system block.

Finally, one other function, **initelm2.m**, is only called when each module is initialized. This function initializes the weights and biases of the network and, in doing so, configures the size of the neural network. **Initelm2.m** is a modification of the original *Neural Network Toolbox* file **initelm.m**. The initialization of the output layer was changed to reflect the fact that the layer transfer function was changed from **purelin.m** to **logsig.m**.

5.5.2.3 Obstacle Avoidance Module

5.5.2.3.1 General

This section provides details of how the sensor input is represented for the obstacle avoidance module as well as how the reinforcement program is written to get the module to converge to the desired behaviour. The m-files **preproo.m** and **rpobst.m** are responsible for these functions and are described next.

5.5.2.3.2 Sensor Input Representation: prepoo.m

The obstacle avoidance module uses the data from Khepera's eight proximity sensors. Each sensor registers a discrete value in the range [0 1024]. In general, the greater the amount of sensory information, the greater the dimensionality of the resulting search space. It is therefore advantageous to choose the coarsest granularity (which still allows a learning architecture to distinguish among the differing states in its environment) in order to keep the search space manageable. The function **preproo.m** reduces the readings from each proximity sensor to a binary condition by thresholding the values, thereby reducing the search space considerably.

In **preproo.m**, an initial thresholding filters out random ambient IR which tended to be prevalent below a sensor reading of about 100 in the environment under study. A second thresholding level of 400 was chosen from empirical measurements so that any objects within about 1 cm became significant. This gave the learning system time to adjust for momentum, slippage, and the delay imposed by the granularity of the motion. Any values above 400 where hard limited to 10 and any below were hard limited to zero. The thresholding levels for a typical proximity sensor response are shown in Figure 5-5. This response represents an approach by the Khepera to an obstacle from a starting distance of 5 cm away. Note that the response curve is quite steep and depends to some extent on the infra-red reflection characteristics of the obstacle.

The maximum value of each sensor reading was hard limited to ten in order to develop reasonably sized activations in the network hidden layer without causing premature saturation of the neurons in that layer. The subject of premature saturation is discussed in Chapter 6.

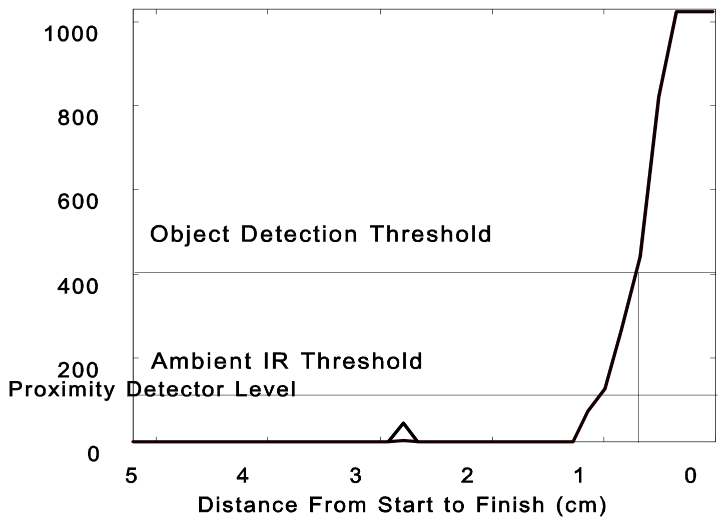

Figure 5-5
Typical Proximity Sensor Response
as Distance to Obstacle Decreases

5.5.2.3.3 Reinforcement Program: rpobst.m

The **rpobst.m** function uses the same array of proximity sensors as does the learning system to determine whether the last action performed was worth rewarding, punishing, or neither. There are no other sensors or sources of data on the Khepera base unit that can provide additional information for this assessment.

In the **rpobst.m** function, the relative merit of the last action is decided by reading the current value of the proximity sensors and evaluating the set of IF...THEN conditions listed in Listing 5-1.

Having determined whether a reward or punishment was given from the conditions above, the values of the target vector, *T*, and the learning rate *LR* are set appropriately for passage to **trnelm1p.m**.

- IF the last action performed was a forward movement AND the values of the front four sensors prior to the action execution were all less than 10, THEN there was obviously nothing wrong with choosing to move forward. This would normally warrant a reward. However, to prevent over-training, the network output *S* is subtracted from the binary valued output *O* (both of which are arguments to the function) and IF the difference is less than a certain threshold, *thresh*, THEN no reward is given since the network is deemed to be sufficiently trained on that action. Otherwise, a reward is given. Note that moving forward is the default behaviour in the absence of any obstacles.

- IF the last action performed was a forward movement AND any of the front four sensors prior to the action were equal to 10, THEN the robot moved into an obstacle. A punishment is given.

- IF the last action performed was a right turn AND the right turn resulted in the sum of the current values of the left 45 and 90 degree sensors being greater than the sum of the current values of the right 45 and 90 degree sensors, THEN the move should be rewarded. That is, a movement to the right should reduce the sum of the proximity sensors on the right in relation to the sum on the left. A check is done against *thresh* to see if the network has learned this behaviour already and, if not, a reward is given. On the other hand, IF the turn to the right results in the sum of the current values of the left 45 and 90 degree sensors being less than those on the right, THEN the robot is treated as having turned inwards towards an obstacle and a punishment is given. Finally, IF there was nothing on the left hand side to begin with, THEN a punishment is given for spontaneously turning right.

- IF the last action performed was a left turn, THEN the set of conditions described in the paragraph above apply except that the right and left sides are switched.

- IF the last action performed was to move backwards AND the previous value of all of the sensors on the front of the Khepera was 10 (an object completely blocking forward motion) then a reward is given. Otherwise, the action is punished.

Listing 5-1
Reinforcement Program for Obstacle Avoidance

The series of conditions above were developed incrementally and were sufficient to cause the desired obstacle avoidance behaviour to emerge. However, the code is as long as that required to directly program the behaviour which was not intended. Elaboration on this issue is given in the *Discussion* in chapter 8.

5.5.2.4 Light Seeking/Approaching Module

5.5.2.4.1 General

This section provides details of how the sensor input is represented for the light seeking and approaching module as well as how the reinforcement program is written to get the module to converge to the desired behaviour. The m-files **preprol.m** and **rplight.m** are responsible for these functions and are described next.

5.5.2.4.2 Sensor Input Representation: preprol.m

For the light seeking behaviour, the sensors were again deemed to be too fine grained. Each sensor produces a discrete value between 512 and 50 (the numbers decrease with intensity). However, in order to get the desired light seeking behaviour to emerge, we were interested in the location of the minimum light value (that is, which sensor it is on) rather than the actual magnitude of the measurement produced by the sensors. That is, if the minimum reading appeared on the right 90! sensor, it can be concluded that the light source is directly in front of this sensor.

The function **preprol.m** determines the sensor having the minimum light reading and sets the value reported by this sensor to 10 while all others are set to zero. The primary reason for this representation was, of course, to reduce the search space. However, the choice of setting the minimum value to 10 and all others zero was not arbitrary. Recall from section 5.5.2.3.2 that the input representation for the obstacle avoidance module set any sensor reading above 400 to 10 while setting all others to zero. As a result, the input vector presented to the network would consist of an 8! 1 vector containing any number of 10's or zeros depending on the current position of the Khepera in relation to its surroundings. Also recall that the value of 10 was chosen in order to develop reasonably sized activity levels in the hidden layer neurons without causing premature saturation. By setting the input representation of the light seeking module to being essentially the same as that of the obstacle avoidance module, we can save time by reusing the network structure (ie number of hidden neurons, learning rates, etc) used for obstacle avoidance. In fact, the state space of the light seeking task is a subset of the state space for obstacle avoidance since only a single value of 10 will be present in any input vector.[1] As a result, any network structure that can perform suitably for the obstacle avoidance task should also do well with the light seeking task.

5.5.2.4.3 Reinforcement Program: rplight.m

The **rplight.m** function uses the same light sensor array as does the learning system to determine whether the last action performed was worth rewarding, punishing, or neither. There are no other sensors or sources of data on the Khepera base unit that can provide additional information. The **rplight.m** function must also be written to train the network on a sequential pair of actions: turn so that the robot is facing a light source and then move towards it.

In the **rplight.m** function, the relative merit of the last action is decided by reading the

[1]In actuality, if two sensors read the same value, then both will be assigned a value of 10. Thus, it is possible to have a vector with multiple occurances of the value 10 - albeit rarely.

current value of the light sensors and evaluating the set of IF...THEN conditions listed in Listing 5-2.

- IF the action just performed was move forward AND either of the front two sensors have a value of 10 (that is, the minimum light value appears on those two sensors) then a reward is awarded. As with the obstacle avoidance behaviour, to prevent over-training, the network output *S* is subtracted from the binary valued output *O* (both of which are arguments to the function) and IF the difference is less than a certain threshold, *thresh*, THEN no reward is given since the network is deemed to be sufficiently trained in that action. Otherwise the network is rewarded. This condition will train the network to move towards a light source when the Khepera is facing it.

- IF the action just performed was move forward AND neither of the front two sensors have a value of 10, THEN the network is punished.

- IF the action just performed was turn right AND the minimum now appears on the back right, right 90, right 45, or front 10 degree sensors, THEN the action is rewarded. A right turn can only result in the light minimum being rotated to these sensors if the light minimum was already on those sensors or if it was on the back left sensor prior to the motion. Hence a reward would be given for rotating the light minimum towards the front two sensors. A check is done against *thresh* to see if the network has learned this behaviour already and, if not, a reward is given. Otherwise, IF the right turn put the minimum elsewhere on the robot's sensors, THEN the network is punished. This will basically punish the network for rotating the robot past the point where it faces the light.

- IF the action just performed was turn left THEN the set of conditions outlined above for the right turn case applies except that the left side sensors are now swapped for the right.

- IF the action just performed was backup THEN punish. This action is to be discouraged at all times as a possible solution.

Listing 5-2
Reinforcement Program for Light Seeking

Having determined whether a reward or punishment was given from the conditions above, the values of the target vector, *T*, and the learning rate *LR* are set appropriately for passage to **trnelm1p.m**.

5.5.2.5 Controller Architecture Design and Implementation

5.5.2.5.1 Introduction

In this section, the design of the controller architecture (Figure 5-1) which integrates the `seeklite` and `avoidobst` behaviours is described. Some changes were required to remove the trainer portion of each behaviour module since training has been frozen. These modifications are described next. This is followed by an analysis of the design of the integration function or *switch architecture* that was used to coordinate the behaviours.

5.5.2.5.2 Modifications to the Behaviour Module

When a network was trained to behave as desired by the reinforcement program, then the trainer section of Figure 5-3 was no longer required. The trained network could then be described completely by its set of weights and biases. To incorporate the weights and biases into the controller design, only the functional blocks shown in Figure 5-6 were required.

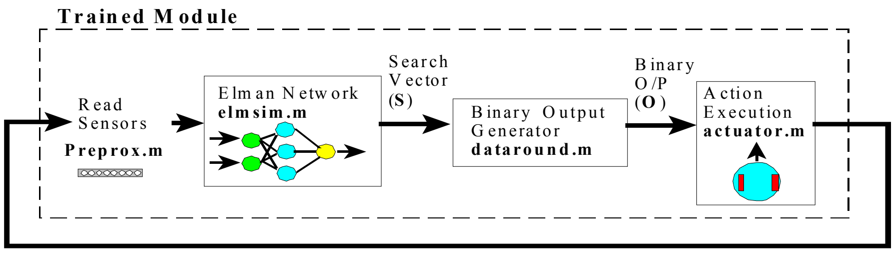

Figure 5-6
Trained Module Architecture

The trained network must still have its inputs preprocessed by the appropriate **preprox.m** function. The network puts out real valued data between 0 and 1 and the switch architecture required binary data as inputs. Therefore the trained network output was passed through the **datround.m** function which hard limits the output using a threshold of 0.5. The outputs are hard-limited because, once the training is frozen, exploration should stop.

5.5.2.5.3 **Module Output Integration Function**

The module output integration function performed a switching operation. In essence, the light seeking behaviour is to have control at all times unless the robot is threatening to run into an obstacle. The integration function knows when to switch control to the obstacle avoidance module because it has access to the readings of the proximity sensors. Although this access is not indicated in the diagram, it is required since both the obstacle avoidance and light following behaviours will only *suggest* a course of action with every state. The integration function needs some other form of information in order to determine *when* each module should have control. This is provided by reading the proximity sensors and producing a binary assessment of whether any is active. If any proximity sensor is active, then obstacle avoidance should control the motion of the robot.

In the test cases provided by the authors of the BAT methodology, particularly the first test case with AutonoMouse V, two individual modules were coordinated via a switch architecture which was itself trained to learn the coordination.

In this work, no apparent gain could be seen by using a neural network to learn the coordination function over using a simple “hard wired” switch that reacted simply to the presence or absence of an obstacle. Accordingly, a simple switch was used as the integration function. The switch gave control to the obstacle avoidance module whenever any proximity sensor indicated the presence of an object or wall.

A co-ordination module could have been implemented by feeding the action votes from each of the behaviour modules to a neural network. The trainer would, of course, require access to the proximity sensors but the neural network would not. If an obstacle appeared on any sensor, then the action vote of the obstacle avoidance module would be mapped to the network outputs by the trainer.