Chapter 2

Background

2.1 Introduction

This chapter begins with a brief outline of the stages of the Behaviour Analysis and Training Methodology. The reinforcement learning paradigm is then introduced, followed by a differentiation between immediate and delayed reinforcement. Elman Simple Recurrent Neural Networks are then discussed, followed by a description of the training algorithm used to train SRNs in this work. A final section is provided describing a method of speeding up the learning in a Simple Recurrent Network using a modified form of the training algorithm already presented.

2.2 The Behaviour Analysis and Training Methodology

This thesis uses the Behaviour Analysis and Training Methodology as a guideline for the engineering design of a robot control algorithm which must seek and approach light sources while avoiding obstacles[1]. The stages of this methodology and a short description of each is given below.

a. *Application description and requirements*: this stage describes the **robot shell** and the initial operational environment of the robot (prior to possible

[1]The light seeking and obstacle avoidance tasks were selected because they offered clearly identifiable and conflicting tasks.

modification). The term **robot shell** refers to the overall "anatomy" of the robot without a complete description of any sensors the robot may have. This stage also requires a formal, quantitative description of the desired overall robot behaviour.

b. *Behaviour analysis*: at this stage, the behaviour requirements specified in the first stage are decomposed into simple structured behaviours (as defined in the methodology) which, when combined, constitute the more complex desired behaviour.

c. *Specification*: this stage determines what must be added to the environment to make the target behaviour possible (i.e. external sensors, colour coded markers, positioning beacons). This stage also establishes the organization of the software controller architecture (but not necessarily how the individual modules of the architecture are to be implemented) and it also identifies the strategy for training the controller. This methodology assumes that the machine learning paradigm will be reinforcement learning and that the reinforcements are immediate (see section 2.3).

d. *Design implementation and verification*: this stage requires the design, implementation, and testing of the robot shell prior to training.

e. *Training*: this stage concerns itself with the training of the robot control architecture according to the training strategy identified in the specification stage. A simulation environment may be used to evaluate the chosen architecture and to develop a first approximation of the controller, thereby speeding up training. The

developed architecture could then be implemented on the physical robot for further training and assessment.

f. *Behaviour assessment*: this stage is reserved for evaluating the training of the controller architecture and for evaluating the observed behaviour.

In this work, the fourth stage, Design Implementation and Verification, is not performed since the Khepera robot being used has already been designed, implemented, and tested.

2.3 Reinforcement Learning Model

The Behaviour Analysis and Training Methodology breaks a complex behaviour up into a series of simpler behaviours. Each behaviour is then implemented by a separate module. These modules are trained to perform the behaviour using machine learning techniques. Specifically, the BAT methodology assumes that the learning system chosen to implement each behaviour module will be based on reinforcement learning.

A generic reinforcement learning model is shown in Figure 2-1. During each cycle of interaction with its environment, the robot receives an input, *i*, through its sensors. This input is an indication of the current state, *x*, of the environment. Based on this input, the robot chooses an action, *a*, which is sent out through the robot's actuators. This action changes the state of the environment, and the *value* of this state transition is communicated to the robot through a scalar *reinforcement signal*, *r*, generated by a teacher or critic.

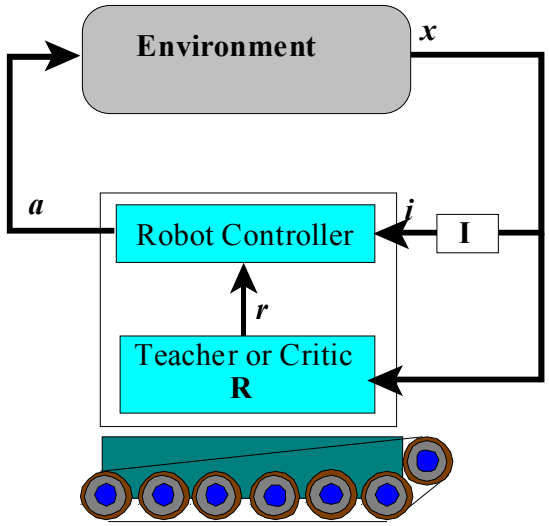

Figure 2-1
Generic Reinforcement Learning Model

Notice in Figure 2-1 that the sensor input may be preprocessed by an input function, **I**, which determines how the robot actually "sees" the environment. If this function were an identity, the robot would perceive the exact state of the environment as presented by its sensors.

2.4 Learning from Delayed Reinforcements

In Figure 2-1, the teacher or critic interprets the current state, *x*, and provides a scalar reward (or *reinforcement*), *r*. If this reward is provided infrequently (ie upon reaching the goal) then the reinforcement is said to be *delayed*. The reinforcement value is used by the robot controller to modify its decision *policy* (ie, the way in which actions are chosen) according to some learning algorithm. Actions executed by the robot affect all successor states and therefore affect all future reinforcement values. The objective of learning is to construct an action decision policy that will maximize reinforcement values received by the robot over the *long term*.

The difficulty of learning from delayed reinforcement is that the robot is never told what actions would be better in particular environmental states from those which it actually performs. Also, it is never told which of the actions that it performed in the past have influenced the reinforcements it received nor by how much. These issues conspire to make the learning task challenging.

2.4.1 Mathematical Framework for Delayed Reinforcement

Consider the learning system interacting with an environment described by a finite set of states X. After observing the state x_t at time t (where x_t! X), the learning system performs an action a_t! A where A is a finite set of actions and is the same for all states X. The action a_t causes the system to make a transition from state x_t to y. This state transition depends only the current state x_t and action a_t and is therefore independent of all previous states of the system. After taking action a_t, the learning system receives a reinforcement r_{t+1} which is also sometimes referred to as a *payoff*. This sequence of actions continues indefinitely.

The objective of reinforcement learning is to find a policy, π, for action selection that is optimal. An optimal policy is usually defined as one which maximizes the expected positive reinforcement, or payoff, received over a given period of time. We concern ourselves here with *stationary* policies. A *stationary* policy specifies actions based only on the current state of the system. The action specified by policy π in state x_t is denoted $\pi(x_t)$. Hence, the system state changes according to:

$$\text{Prob}\{x_{t+1}=y|x_0,a_0,x_1,a_1,...x_t=x,a_t=a\}=\text{Prob}\{x_{t+1}=y|x_t=x,a_t=a\}=P_{xy}(a)$$

where $P_{xy}(a)$ is the probability of moving from state *x* to state *y* and depends on the action taken. A sequence of states defined as above forms a first-order Markov chain with transition probabilities $P_{xy}(a)$.

A natural and popular measure of a learning system's performance is known as the

$$V^{\pi}(x) = E_{\pi}\left[\sum_{t=0}^{\infty} \gamma^{t} r_{t+1} \mid x_0 = x\right]$$

evaluation (or value) function $V^{\pi}(x)$ and is defined as follows [Barto89]:

(2.1)

The value function above is based on the *infinite horizon discounted model* of optimal behaviour. The summation inside the expectation is called the *cumulative discounted reinforcement*. The term r_{t+1} is the reinforcement received from the environment after action a_t is taken by the learning system. The factor γ (where $0<\gamma<1$) is called the *discount-rate parameter*. By adjusting γ, it is possible to control the extent to which the learning system is concerned with long-term versus short-term consequences of actions.

In each state, V(x) *represents* the expected cumulative return over the infinite number of time steps from t=0 assuming that the systems starts in state x and uses policy π to determine all future actions. Since the system is stationary (and future behaviour does not depend on how the system arrives at the current state), then the evaluation function is, in fact, the expected return

beginning *whenever* the system enters any state under the condition that action selection policy π is continued.

The evaluation of a state is a *prediction* of the return that will accrue throughout the future whenever this state is encountered. The basic idea behind reinforcement learning is to learn the evaluation function V(x) so as to predict the cumulative discounted reinforcement to be received in the future from any state *x*.

2.4.2 AHC and Q-Learning

The Adaptive Heuristic Critic ([Barto83], [Sutton84]) and Q-learning ([Watkins92], [Humphrys95], [Lin93b]) are two popular and well studied examples of algorithms which are capable of learning from delayed reinforcement. They are both based on the idea of learning an evaluation function to predict the discounted cumulative reinforcement to be received in the future. They differ in the following respect [Lin93a]:

- The Adaptive Heuristic Critic (or AHC) learns a V function. The V function represents the expected discounted cumulative reinforcement that will be received starting from world state *x* as defined previously.

- Q learning learns a Q function. The Q function, Q(x,a), represents the expected discounted cumulative reinforcement that will be received after the execution of action *a* in response to world state *x*. It is the *utility* of performing state-action pair (x,a) as opposed to just the value of a state *x*.

Both algorithms learn the evaluation function using the method of temporal difference (TD) learning [Sutton88]. In essence, TD learning computes the error between temporally successive values or predictions and some learning architecture (traditionally neural networks) minimizes this error over time. For example, expanding the summation in (2.1), we obtain

$$V(x_t)= r_1+\gamma r_2 +\gamma^2 r_3...$$

$$= r_t+\gamma V(y_{t+1})$$

$V(x_t)$ represents the expected cumulative discounted return for state *x* at time *t* and $V(y_{t+1})$ represents the expected return for state *y* at time *t+1*. r_t represents the reinforcement value actually received for moving from state *x* to state *y*. The *difference* in temporally successive predictions is then computed as:

$$\Delta= r_t+\gamma V(y_{t+1})-V(x_t)$$

This difference provides a gradient of *secondary reinforcement* by anticipating the events that provide primary reinforcement (ie from a teacher or the environment). This error is minimized by some learning system over time so that the evaluation of state x_t is drawn towards $r_t+\gamma V_t(x_{t+1})$. That is,

$$V^{\pi}(x_t) \equiv \sum_{t=0}^{\infty} \gamma^k r_{t+k}$$

This process is conceptually similar to applying one step of the successive approximations method of computing a value function as specified by the following principle

equation in dynamic programming:

$$V^{\pi}(x_t) = R(x_t, a_t) + \gamma \sum_{y} P_{xy}(a_t) V^{\pi}(y)$$

In both the Adaptive Heuristic Critic and Q-Learning, the reinforcement actually received at step t+1, r_{t+1}, is substituted for the expected value of the reinforcement $R(x_t,a_t)$. Since $E[r_{t+1}|x_t,a_t] = R(x_t,a_t)$, r_{t+1} is an unbiased estimate of $R(x_t,a_t)$. Additionally, $V(y_{t+1})$ is substituted for $\sum_y P_{xy}(a_t)V(y)$. $V(y_{t+1})$ is the evaluation estimate of the next system state using the current parameter values of the learning system. Although this substitution does not necessarily result in an unbiased estimate of the desired quantity, it is usually assumed that the approximation is satisfactory.

2.4.3 Advantages and Disadvantages of Delayed Reinforcement

The great advantage of using delayed reinforcement techniques is that it is sufficient to provide a sensor that indicates only when a goal state has been reached. It is up to the controller to determine the optimality of the chosen path to the goal. However, actions cannot be evaluated unless they are actually performed. To obtain a very close approximation of the value function, and in the absence of a good generalization method (ie lookup tables – which offer no generalization – are used to store knowledge), then every state in the environment must be visited a large number of times in order to develop the approximation. Invariably, this means that systems based on delayed reinforcement tend to converge very slowly since the learning system must perform many thousands of trials. Methods of speeding up the convergence to

optimal behaviour remains an active field of research (eg [Lin93a],[Thrun92],[Gullapalli92]).

2.5 Learning From Immediate Reinforcements

Immediate reinforcement occurs when the teacher or critic provides a scalar evaluation of the action that was just executed at *every* iteration of the control algorithm. As a result, the robot control algorithm is no longer responsible for determining the long term consequences of actions being performed. This task is passed on to the teacher.

The provision of a teacher or *reinforcement function* is natural. As in real life, there is no reason not to exploit the knowledge of a teacher to achieve faster learning. Indeed, technological progress would invariably grind to a halt if every student was required to re-learn everything from scratch on his or her own.

The price paid for this expediency is that the reinforcement function must be provided with a greater quantity of information which may be more difficult to obtain. For instance, Columbetti et al. [Columbetti96] cite an example where the reinforcement function is required to compute the distance between the robot and some goal position at every iteration in order to generate suitable reinforcements. It is no longer sufficient to endow the learning system with a sensor to indicate the achievement of the goal state. Gathering and interpreting information at every iteration and using it to provide suitable reinforcement can be problematic and expensive in terms of hardware.

The Behaviour Analysis and Training Methodology is based on the concept of *Robot Shaping* [Dorigo94]. This concept relies on immediate reinforcement learning as opposed to delayed reinforcement learning. In fact, the reinforcement function, **R**, in Figure 2-1 becomes

what the authors term the "trainer." Given that this is the case, the main problem of teaching is to ensure that the right thing is being taught and this means that the *most critical component in the methodology is the trainer*, or reinforcement function, **R**.

2.6 Feed-Forward and Simple Recurrent Neural Networks

In this thesis, Elman Simple Recurrent Networks were used as the learning system for each behaviour module. Since Elman SRN's are a type of feed-forward neural network, a description of these follows immediately. This, in turn, is followed by a description of the structure of an Elman SRN.

2.6.1 Feed Forward Neural Networks

A typical feed-forward neural network is shown in Figure 2-2. Neural networks consist of many very simple processing units called neurons which communicate by sending signals to each other over a large number of weighted connections [Haykin94]. These weighted connections provide neural networks with a propensity for storing large amounts of experiential knowledge through training.

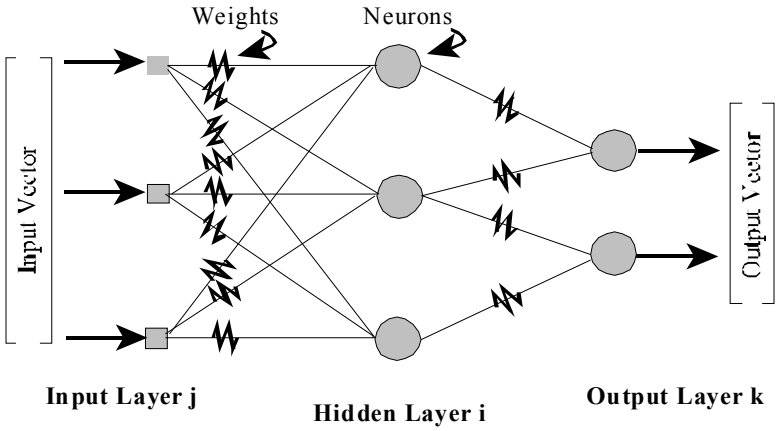

Figure 2-2
Typical Feed-Forward Neural Network

In general, a neural network has the layered structure shown in Figure 2-2. Each layer consists of units which receive their input from units in the previous layer. The input layer receives its values directly from the environment. A fully connected neural network (as shown in Figure 2-2) is one in which the outputs from every unit in a particular layer are fed to every other unit in the next layer. The *hidden layer* is aptly named to reflect the fact that it exists exclusively between the input layer and the network output layer. There may be any number of hidden layers.

2.6.1.1 Neural Network Training: Forward Propagation

A neural network can be trained using the *back-propagation algorithm* [Rumelhart86]. Using this algorithm, the error between the values presented by the network at the output layer and some desired set of values is minimized over a large set of training data. This is done in two steps. The first step is the *forward propagation*, in which a training "sample" (typically a *vector*)

is presented at the input layer of the network[2]. The *activation*, or output, of each neuron is computed in a layer-by-layer fashion. The activations from each layer are fed to the next layer until the output layer is reached.

The activations of each neuron in the hidden and output layers are computed as follows. First, the activity level v_i at the input to neuron *i* in the current layer is computed as

$$v_i = \sum_j w_{ji} y_j + b_i$$

where $\sum w_{ji}y_j$ is the weighted sum of the activations from every neuron in the previous layer (y_j is just the input vector if the previous layer is the input layer). The term b_i is a bias indigenous to each neuron. This bias term is typically considered as the strength of a connection from a unit with a constant activity level of 1.

The activation of each of the neurons in the hidden and output layer is computed from the activity level v_i by passing it through an *activation function* F_i. That is,

$$y_i = F_i(v_i)$$

[2]Note that, although the input layer is termed "layer", it does not actually contain any neurons. The *activation levels* from the input layer will simply be the input signals.

In general, the activation function (also known as a *squashing function* or *transfer function*) is non-linear and its purpose is to limit the amplitude of the output of a neuron. Examples of commonly used activation functions include differentiable sigmoid functions. The overall computation of the activation level of a typical neuron in either the hidden or output layer is summarized in Figure 2-3.

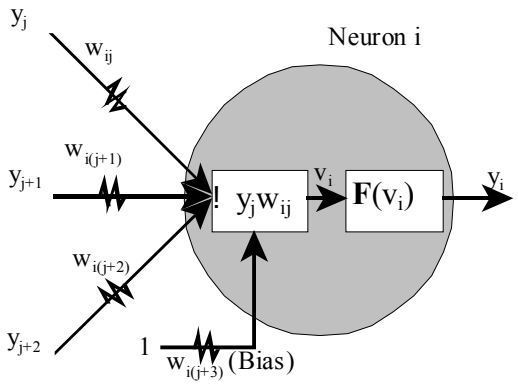

Figure 2-3
Typical Neuron Activation Computation

As shown in the next section, it is required that the activation function be differentiable everywhere since it's derivative must be computed. In this work, we chose to use a hyperbolic tangent sigmoid as the activation function because this transfer function is very commonly used in the hidden layer of multi-layer neural networks. It is defined as:

$$F(V_i) = \frac{1 - e^{-V_i}}{1 + e^{-V_i}}$$

The hyperbolic sigmoid transfer function maps the neuron input from the interval (-! ,!) to the

interval (-1,1).

Since the output layer neurons were required to generate real values on the range (0,+1), the output layer activation function in this work was a log sigmoid, defined as:

$$F(V_i) = \frac{1}{1+e^{-V_i}}$$

This function maps the neuron input from the interval (-! ,!) to the interval (0,+1).

2.6.1.2 Neural Network Training: Back-propagation

Once the activations have been computed for each layer in a forward direction, the activations appearing on the output layer represent the network response to the input vector. During the forward propagation, all weights and biases were held constant. A *backward propagation* (or *back-propagation*) is then performed to adjust the weights and biases of the network. Specifically, the actual response of the network is subtracted from some desired or target response to produce a vector of error signals. This vector is then propagated backward through the network and the weights and biases are adjusted according to some learning rule so as to make the actual response of the network move closer to the desired response.

The *instantaneous* sum of squared errors of a neural network is a function of all the free parameters (weights and biases) of the network and is written

$$! = \frac{1}{2} \sum_{i} (d_i - y_i)^2 !$$

where $e_i=d_i-y_i$ represents the error, between the desired response d_i and the network response y_i and the set C includes all the neurons in the output layer of the network. By using gradient descent, we may modify the weights iteratively to minimize ! over a set of training vectors. That is, the back-propagation algorithm applies a correction Δw_{ji} to the weight w_{ji} (from neuron *j* in the previous layer to neuron *i* in the current layer) which is proportional to the instantaneous gradient ! ! /! w_{ji}. The correction Δw_{ji} applied to w_{ji} is defined by the *delta rule* as

$$\Delta w_{ji}=-\eta\frac{\partial \mathcal{E}}{\partial w_{ji}}=\eta\delta_i y_j$$

where η is a constant which determines the rate of learning. For an output neuron, the *local gradient* δ_i is computed as

$$\delta_i=(d_i-y_i)\,F'(V_i)$$

For a neuron in the hidden layer, δ_i is computed by

$$\delta_i=F'(V_i)\sum_k \delta_k w_{ki}$$

where δ_k represents the local gradient of neuron k of the layer to the right of the current layer and the summation is taken over all neurons in that layer. Note that the back-propagation algorithm

requires that the activation function be differentiable everywhere since its derivative appears in the local gradient computation.

2.6.2 Elman Simple Recurrent Networks

The structure and training of Elman Simple Recurrent Networks do not differ substantially from that of a feed-forward neural network. A simplified diagram of an SRN is shown in Figure 2-4 (bold arrows indicate the full interconnection between layers).

In an SRN, an additional bank of units is added at the input layer. These units are labelled the *context units* [Elman90] and they represent a *copy* of the activation levels from each of the neurons in the hidden layer. Consequently, the context layer will have the same dimension as the hidden layer. Although they appear in the input layer, the context units are really hidden units in the sense that they interact exclusively with other neurons in the network and not the outside world.

Training of an Elman SRN using the back-propagation algorithm proceeds in exactly the same manner as described for a regular feed-forward neural network with the following exceptions. First, the activation levels of each neuron in the context layer are set to some initial value for the very first forward propagation of the very first training vector. In this work, the initial values were set arbitrarily to zero because the same initial values must be used for each trial and large values could potentially saturate the hidden layer neurons (see Chapter 6). Second, after each back-propagation phase, the activation levels of the hidden layer are copied to the context layer. Effectively, the context layer reflects the state of the hidden layer delayed by one sample cycle. This is illustrated as the box containing the D operator in Figure 2-4.

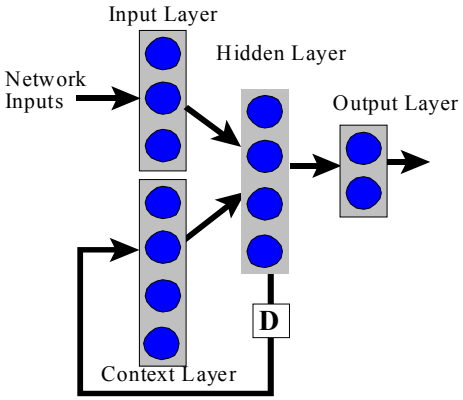

Figure 2-4
Elman Simple Recurrent Network

As Elman explains, the hidden units of a normal feed-forward network develop internal representations for the input patterns as the network is trained. These representations enable the layer to recode the input patterns in a way which enables the network to produce the correct output for a given input after sufficient training.

For a simple recurrent network, the context units are charged with remembering the previous internal state of the hidden layer and presenting this state to the hidden layer on the next time step. Thus, the hidden layer faces the task of mapping both an external input as well as its own previous state for some desired output. The hidden layer must accomplish this mapping and at the same time develop representations which are useful encodings of the temporal properties of the sequential input.

The internal representations that develop are sensitive to temporal context. Effectively, the network develops a memory that is useful for recognizing sequences of input vectors. More importantly, the effect of time is implicit in these internal states, eliminating the task of

determining how a memory is to be designed into the system. The memory is highly task and stimulus dependent.

2.6.3 Advantages of Neural Networks for Knowledge Representation

Besides their temporal processing capabilities, recurrent neural networks inherit several advantages for use as a knowledge representation architecture by simply belonging to the class of *connectionist* systems.

First, multi-layer neural networks scale well with increasing input size. Lookup tables, Radial Basis Function Networks, and CMAC's (Cerebellar Model Articulation Controllers) have all been used as the means of storing knowledge in learning systems. However, the number of individual basis functions or entries in a lookup table or CMAC rises exponentially with the dimension of the input vector so that these storage systems end up suffering from the curse of dimensionality. That is, they do not scale well with increasing dimension.

By their very nature, connectionist architectures do not need to represent explicitly all possible situation-action combinations since they display good generalization capabilities and therefore do not fall prey to the curse of dimensionality. They can typically respond appropriately to novel instances which are similar to trained examples. The generalization property also allows them to operate with continuous values – which is a big advantage over many other knowledge storage and retrieval systems.

Neural networks tend to deal well with noisy input data which is essential for a robot working in the real world (eg [Millan96]) and contributes greatly to the robustness of a learning system based on them. Robustness is also elevated by the fact that, since there is a large number

of simple processing units, each with primarily local connections, damage to a few of these need not impair overall performance significantly.

Finally, neural networks can accept and produce vector valued quantities. Their structure is amenable to fast parallel processing and implementation in Very Large Scale Implementation technologies.

2.7 Network Training Algorithm

In this work, the back-propagation algorithm is used to train the SRN's. However, as discussed in the next section, the fact that the network produces a vector valued output but only receives a scalar evaluation of its performance leads to a *structural credit assignment* problem. The solution, Complementary Reinforcement Back-Propagation, is described in Section 2.7.2.

2.7.1 The Structural Credit Assignment Problem

In a supervised learning context, a neural network would produce a vector of outputs in response to a certain input pattern. This vector would be subtracted from a target output vector. The difference produces an error vector that is used to adapt the weights and biases of the network such that a future occurrence of the same input pattern produces an output vector which is closer to the supplied target vector. This type of learning is *directed* [Barto92]. The agent is explicitly told how to change its behaviour to improve performance.

However, the networks in each of the behaviour modules are trained in an *evaluative* fashion and do not receive an explicit target vector. Instead, they receive a scalar reinforcement which may take one of three discrete states: -1, 0, or 1. This scalar must be used in some manner by the network to adjust its weights and biases but the scalar does not provide any

specific information concerning the error gradient of each output. That is, an error vector is not constructed. This creates a *structural credit assignment problem*. Gradient descent, the method by which a neural network solves the credit assignment problem in a supervised learning environment, is not possible without an error gradient vector. The network training algorithm must somehow take this credit assignment problem into consideration.

2.7.2 Complementary Reinforcement Back Propagation

The network training algorithm used in this work is Complementary Reinforcement Back Propagation [Meeden 94]. It solves the problem of structural credit assignment by *deriving* a target vector for the network with which to compute an error gradient. It does this in the following manner. First, on the forward propagation of the sensor input, the network output layer generates a search vector S whose elements are in the range (0 1). Next, a binary output or *action* vector A is created from the search vector S by generating a random number[3] for each element in S. If the random number computed for element i in S is less than the corresponding value of element i in S, then element i of A is set to 1 and zero otherwise.

The binary action vector A is used as an action selector as well as to provide the error gradient for back-propagation (and thereby provide a solution to the credit problem). If the action selected by A receives a reinforcement value of 1 (ie, a positive reward), then the network is trained on the error vector (A-S). This pushes the network output S closer to A. If the network

[3]The random numbers are chosen from a uniform distribution on [0 1] in [Meeden94]

receives a reinforcement value of -1 (ie, a negative reward, or punishment), the network is trained on the complement of action *A*, or ((1-*A*)-*S*). This pushes the network output *S* in a direction opposite from *A*.

Although it may appear that the algorithm will begin oscillating between an action and its complement, this does not happen. A single step of the algorithm will only change the action slightly and since the search vector *S* tends to move towards 0.5, the action selection becomes more random. The random selection of the binary output vector will, at some point, generate an action that works better, and then that action will be reinforced. The stochastic selection of the output vector provides the network with an *exploration* capability.

In the fashion described above, action outputs that have been rewarded are more likely to occur, while action outputs that have been punished will tend to move towards the complement action output in similar situations. Note that the rewarding of an action does not necessarily imply that the action is correct or that any significant new learning has taken place. For example, if a network receives a string of consecutive rewards for selecting the same action over and over again, then it is simply driving its network outputs closer to the binary value of the action it is selecting. If the frequency of action selection were doubled for this sequence, it would not really be learning at twice the prior rate. Additionally, if the network were part of the control system of a robot, and the sequence of rewarded actions were driving it towards a cliff, it can be argued that it is not the network but the system providing the rewards that has failed.

To summarize, the steps of the CRBP training algorithm for one full iteration or *cycle* are as follows:

- Get sensor input
- Propagate input through Elman network and obtain search vector *S* on (0 1)
- Stochastically determine binary valued *A* from *S* using uniformly distributed random numbers
- Execute the action corresponding to *A*
- Compute the error gradient: (*A*-*S*) if rewarded, ((1-*A*)-*S*) if punished, zero otherwise
- Back-propagate the error gradient and adjust free parameters of network

2.7.3 Increasing the Learning Speed of CRBP

2.7.3.1 Exploration VS Exploitation and Network Confidence

One major difference between reinforcement learning systems and supervised learning systems is that a reinforcement learner must explore its environment. This brings into play a number of difficult design questions [Thrun92]. For example, *how does one trade-off exploration and exploitation*? Exploration and exploitation are opposing objectives. That is, the environment must be sufficiently explored in order to identify an optimal control policy. However, experience gained during learning must also be considered for action selection if one is interested in minimizing the cost of learning. Integral to this trade-off are questions concerning the design of the exploration rule: *what kind of exploration rule should be used and what impact has the exploration rule on the speed and the costs of learning*?

There are a number of formally justified exploration techniques (eg dynamic programming, Gittins allocation indices, learning automata) as well as some ad-hoc techniques

(greedy strategies, randomized strategies, interval-based techniques). CRBP uses a randomized strategy. The closer to 0.5 a network output bit is, the more random will be the action selection. As the output bits get closer to zero or one, the probability and actual occurrence of the opposite bit decreases.

However, this exploration mechanism was found to be inefficient and lead to longer learning times in this work. The reasoning behind this is that the random strategy used by CRBP is too liberal. It does not make a sharp enough distinction between when to explore and when to exploit. The result is that the learning architecture spends a portion of its time exploring fruitlessly, often being "confused" by the exploration rule.

To illustrate, we introduce the term *confidence* as a description of the learning architecture's conviction of its choice of action. Assume that, in a particular situation, the forward propagation of a sensor state results in the control vector [0.51 0.48] (where each element of the vector represents an output bit of the network). This set of values indicates that the network is not at all confident about the choice of action to take in the current situation. At this point, a random action selection by the output vector is justified.

Now assume that a particular state has been visited a number of times and the control vector generated is [0.90 0.90]. Thus, based on prior experience, the network is 90% confident in its selection of each bit. Using a uniform random distribution, there is a ten percent chance on each bit that the vector selected will not be the vector that the network "thinks" it is selecting. This represents a fundamental disconnect between action selection and action execution that can be responsible for "confusion" in the network.

In the above example, assume that the action vector selected was [1 1] and that this vector was propagated to the actuators. Assuming that this action was rewarded, the network output will be pushed further towards [1 1] (that is, it becomes more confident of the correct action to chose in the given situation). Now assume that the vector was changed to [1 0] by the random exploration function. Regardless of whether the resulting action is rewarded or punished, the network output is still likely to be [1 1] when this state is again visited so that the act of pushing the network towards [1 0] (or [0 1] if punished) really only delays learning by a cycle. Unless the dynamics of the learning environment or the goal of the trainer has changed, choosing [1 1] is still likely to receive a positive reinforcement. Indeed, if both outputs were chosen as [0 0] and this action was punished, the target vector would be [1 1] and the network would, in fact, have never noticed having performed an incorrect action.

2.7.3.2 Gaussian Distributed Random Exploration

It was postulated that using a Gaussian distribution would provide faster learning than when using a uniform distribution of random numbers in CRBP. The Gaussian distribution allows a gradual decrease of the likelihood of random numbers occuring as the network outputs get closer to zero or one (i.e. the network is becoming more confident). The shape of the Gaussian distribution confines the network exploration towards the mean as opposed to uniformly distributing which, as its name suggests, allows unrestricted exploration. Due to it's ease of implementation within Matlab, a Gaussian distribution was evaluated against the uniform distribution. Using the Gaussian distribution, exploration was allowed within a tight region about 0.5 - or when the network is not sure of the correct action in the given state. However,

when an action has been rewarded a number of times, the outputs of the network move away from 0.5 into the region of growing confidence. As a result, exploration decreases sharply. If the dynamics of the environment or the goal of the trainer changes, and the network response is no longer valid in a particular state, it "relearns" by successive punishment. The punishments drive the network in the opposite direction – i.e. back to the 0.5 region – where exploration once again becomes a large factor in action selection.