THE UNIVERSITY OF CALGARY

Reinforcement Learning in Neural Nets

by

Brian Schack

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
OCTOBER, 1988

ISBN 0-315-50381-5

Canada

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Reinforcement Learning in Neural Nets" submitted by Brian Schack in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor,
Dr. Brian Unger
Department of Computer Science

J. B. Hyne, Dean
Faculty of Graduate Studies

Dr. Brian Gaines
Department of Computer Science

Dr. Bruce MacDonald
Department of Computer Science

Dr. Jerry Ells
Department of Psychology

October 24, 1988

Abstract

This thesis describes a learning system called *Extended Self-Propagating Search* (ESPS). ESPS is an extension of *Self-Propagating Search* (SPS), a model of the cerebellum. The extensions enable ESPS to solve reinforcement learning problems, a class of problems unsolvable by SPS. Reinforcement learning problems are characterised by an environment that rates the actions of the learning system through a reinforcement signal. The best solution is the one which receives maximum reinforcement. Solving reinforcement learning problems requires being able to internally generate new solutions, and to determine which portions of a solution are responsible for the reinforcement received. These abilities were added by making data retrieval non-deterministic, and by introducing *expectation*, which expresses ESPS's expectation of future reinforcement. A mechanism for associating expectations with individual steps in a solution and for manipulating those expectations was developed. The changes made to SPS were, like SPS itself, physiologically plausible.

ESPS has been implemented on a network of processors running in parallel, communicating via the Jade interprocess communications facility. ESPS was tested on two problems - the *single-step* problem and the *multi-step* problem, which differ in the frequency of reinforcment. It has successfully solved the single-step problem. Its results are compared to a similar test performed on the Associative Search Network of Barto, Sutton, and Brouwer. ESPS has not solved the multi-step problem. The reasons for this failure are discussed, as is a possible remedy.

Acknowledgements

I could not have completed this thesis without the guiding hand (and foot, sometimes) of Bruce MacDonald. He displayed unflagging enthusiasm in spite of my poor initial attempts, promptly returning draft chapters heavily commented with his distinctive "handwriting". And while we may have at times disagreed on his comments, it was his willingness to take the time to read my work, in spite of being under no obligation to do so, which was most important. For this I am truly grateful.

I would like to thank my supervisor, Brian Unger, for his patience and helpful comments.

I would like to thank the office and support staff for their help, especially in these last hectic months where everything that could go wrong did go wrong. Larry Mellon, Dave Mason, and Earl Locken should be thanked for their rapid responses to my numerous letters about Jipc and ditroff. Bev Frangos was especially helpful in maintaining a stable working environment for me while the rest of the department was being shifted hither and yon. I shall never forget her offer of help during one deep dark day in August when tbl was living up to its horrid reputation.

Anja Haman, Rosanna Heise, and Konrad Slind are to be commended for their bravery in reading early drafts of my thesis.

Most importantly, I would like to thank my friends and family. Had it not been for their company and support, it would not have been possible to survive these last four years, let alone enjoy them as I have. First, to my friends, thanks for the too many games of frisbee, the too many hikes, the too many conversations having nothing to do with computer science, and in general the too many fun ways of not doing a

thesis. Charles Herr and Anja Haman should be singled out as particularly good people with whom not to work on a thesis. I just hope I can waste as much of your time, and in such an enjoyable manner, as you did of mine. I'll have some big shoes to fill.

My family has given me tremendous support over the last four years. They were always there when I needed them. Thank you all.

Table of Contents

CHAPTER 1

Introduction

This thesis outlines a learning system developed by Pentti Kanerva (1984) that models the human cerebellum, extends the system's learning power, and applies the extended model to problems unlearnable by the original model. The original model is Kanerva's (1984) Self-Propagating Search (SPS), and the extension is called Extended Self-Propagating Search (ESPS). Extending SPS involved devising a scheme for internally generating solutions, a scheme for evaluating the effects of a solution, and a scheme for applying the results of the evaluation to generate a new solution. The particular schemes devised were designed to maintain the physiological plausibility of ESPS as a cerebellar model.

1.1. Learning Systems

SPS and ESPS are learning systems. The aim of a learning system is to learn to solve a certain task (i.e., to achieve a certain goal). Example tasks include games such as checkers, chess, and blackjack (the aim being to win), as well as controlling physical systems like robot arms (the aim being, for example, to move the end effector to a certain point in space). An example of a physical learning problem is the *pole-cart problem* (Barto, Sutton, and Anderson, 1983). In the pole-cart problem, a learning system is presented with a two-dimensional cart which can roll between fixed barriers. Motion is produced by applying forces to either end of the cart. On the cart is a pole, hinged at the bottom. The task is to keep the pole balanced by moving the cart (figure 1.1). The learning system is given information at each step on the current angle and angular velocity of the pole, and the position and velocity of

the cart. These together constitute the current *situation.*

Tasks such as the pole-cart problem are complicated enough that they require multi-step solutions. That is, the physical system will start in a certain situation, and on the basis of the current situation, the learning system chooses an action. The action is performed, which results in a new situation, and the learning system chooses a new action to perform. This process is repeated until a goal is reached, or in the case of the pole-cart problem, until the pole falls over or the cart touches the barriers. Therefore, the aim of a learning system is to learn a set of <*situation,action*> pairs that together will solve the task. This paradigm views learning systems as simple production systems, and is the paradigm used by SPS, ESPS, and all the learning systems reviewed in chapter 2, with the exception of Holland's classifier system (Hol-

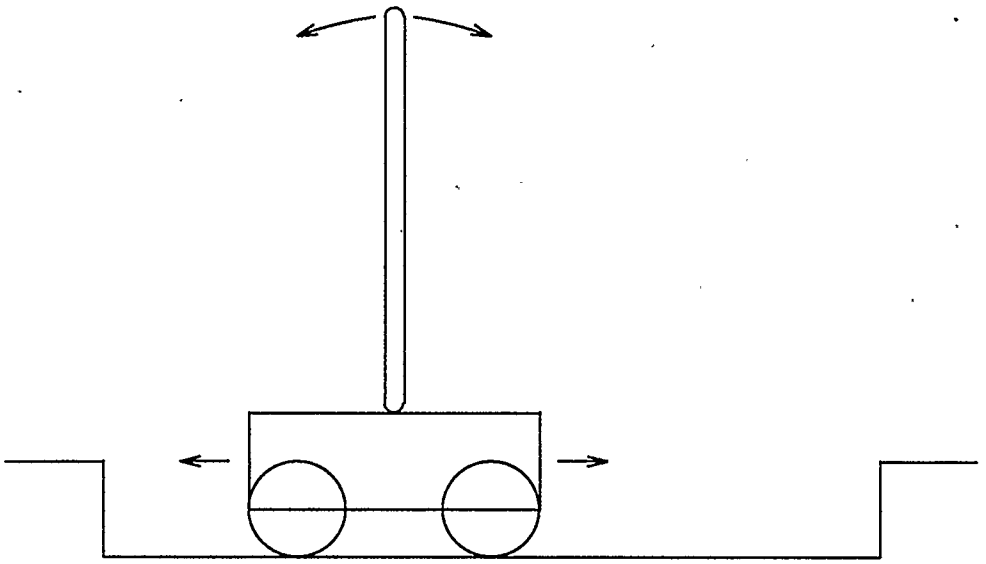


Figure 1.1. The Pole-Cart Problem.

land, 1986) and simulated annealing (Kirkpatrick, Gelatt, and Vecchi, 1983).

Implicit in this formulation of the problem is the assumption that all the information necessary to differentiate two situations must be in *situation*. This is because the choice of an action is made exclusively on the basis of the current situation. No other information comes into play.

SPS belongs to a class of learning systems called *supervised systems*, while ESPS belongs to a class of learning systems called *reinforcement systems* (Barto, Sutton, and Anderson, 1983). Both supervised and reinforcement systems learn <*situation*,*action*> pairs; they differ in how much help they require to learn those pairs, as will be explained in the next two sections.

1.2. Supervised Systems

Supervised systems are found mainly in the field of pattern recognition (Kohonen, 1984, Anderson, 1983, Hopfield, 1982, Albus, 1975, Kanerva, 1984). Pattern recognition problems are characterised by an environment which offers "high quality" feedback: when the learning system produces an incorrect result, the environment gives an immediate and clear indication of the error and how to correct it.

Figure 1.2 illustrates the inputs to and outputs from a supervised system. It takes as input the current situation, and produces as output an action. If the action is incorrect, the environment supplies the correct action (or the difference between the system's action and the correct action). The line marked "r/w" is a read/write signal. In normal operation, it is set to "r". Upon presentation of a situation, the system will produce an action. When the read/write signal is set to "w", the system will pair the current situation with the action supplied by the environment.

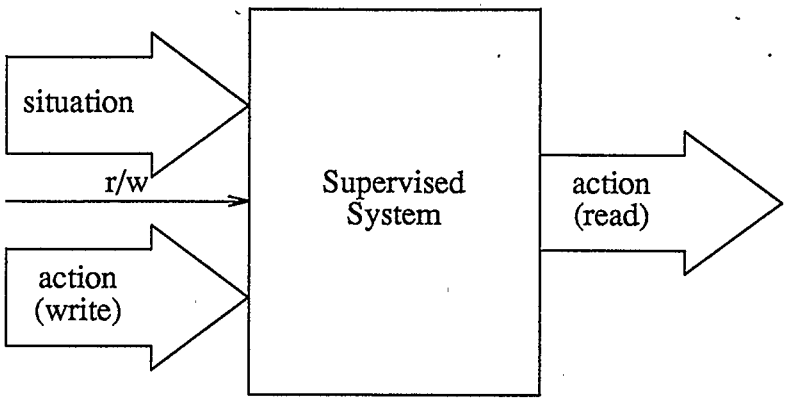


Figure 1.2. Supervised System.

Thus the basic function of a supervised system is to store <*situation*, *action*> pairs. After storage of a <*situation*, *action*> pair, presentation of *situation* will produce *action*.

As will be seen later, only a subset of pairs need be given to many supervised systems, as they can successfully interpolate the rest. Nevertheless, *the environment must supply some correct <situation,action> pairs*. The learning system is not capable of deriving those pairs independently. So essentially the problem must already have been solved, and pairs needed to solve the problem must be supplied to the learning system by the environment. In other words, it requires an environment which is "smarter" than the learning system is.

It is not necessary to talk of supervised systems as storing <*situation,action*> pairs - there are several alternative and equivalent ways of viewing this process, each appropriate to some particular task. One, which was alluded to in the first paragraph of this section, is to view a supervised system as a pattern recognition system. In-

stead of <*situation,action*> pairs, we have <*pattern,class*> pairs (i.e., a pattern and its classification). After storage of a <*pattern,class*> pair, presentation of *pattern* will produce *class* (i.e., *pattern* will be classified). A supervised system can also be viewed as a random access memory. The situation is an address into the memory, and the action is the data stored at that address. Thus, instead of <*situation,action*> pairs, we have <*address,data*> pairs. This view is particularly well illustrated by figure 1.2. Finally, a supervised system can also be viewed as learning a function. The situation then becomes the input to the function, and the action consists of the output of the function. Thus we have <*input,output*> pairs.

That a supervised system can be viewed as a random access memory suggests that constructing a supervised system is trivial. However, a discussion of the further demands usually made on a supervised system will dispel that notion.

First, supervised systems are often expected to recognise novel patterns, classifying them on the basis of similar, stored patterns. That is, assume that the supervised system has learned a set of pairs, $\{<p_0, c_0>, \ldots, <p_{sim}, c_{sim}>, \ldots, <p_n, c_n>\}$, and is presented with a novel pattern, p. If p_{sim} is the stored pattern most similar to p, then the supervised system should return c_{sim}.

Classifying novel patterns in this way is equivalent to solving the *best match problem* (Minsky and Papert, 1969, Kanerva, 1984). The conclusion of Minsky and Papert on the ability of a regular random-access memory to solve the best match problem is pessimistic. They concluded that a memory of size $b_c 2^{b_p}$ is needed, where b_p is the number of bits needed to represent a pattern and b_c is the number of bits needed to represent a class. That is, 2^{b_p} locations are needed (one for each possible pattern), and at each location b_c bits are needed to store the class designation. The memory is primed such that location p contains c_{sim}, where p_{sim} is the most similar

pattern to p. When this is the case, classification can be done by reading at p.

Even disregarding the time needed to prime memory, consideration of typical patterns will quickly reveal this scheme as impractical. Consider, for example, when images are used as patterns. Images are typically on the order of hundreds of thousands of bits in size. Even when the image size is 10×10, with one bit per pixel, the pattern size would be 100 bits, and 2^{100} ($\approx 10^{30}$) memory locations would be required.

Thus, solving the best match problem poses many difficulties. However, once a supervised system can solve the best match problem, many useful abilities naturally emerge. We have seen already that classifying novel situations becomes possible when a system can solve the best match problem. It is also possible to deal with noisy input patterns. The system simply finds the stored pattern, p_{sim}, most similar to the input pattern, p_{noisy}, and returns the class stored with that pattern, c_{sim}. This procedure is unsuccessful when the input pattern, p_{noisy}, is sufficiently corrupted that its best match, p_{sim}, is different than the uncorrupted original pattern, p_{good}. However, when given no information on the corruption that has occurred, as is the case here, the learning system cannot be expected to recreate a badly corrupted pattern. The best we can expect is for it to return its best guess as to the original pattern, and that guess is based on similarity to stored patterns.

If we view the supervised system as storing <*situation*, *action*> pairs, we can see how it deals with novel situations. In a novel situation (a situation that has not been stored), the system will return the action stored with the most similar recorded situation. For tasks with continuous domains (i.e., performing two similar actions in a situation will produce similarly good results), this is a useful property.

Because reinforcement learning systems often work in domains where these abilities are important, supervised systems are often used as the basis for reinforcement learning systems (e.g., see Widrow, Gupta, and Maitra, 1973). This is true of ESPS as well, which uses SPS, a supervised system.

SPS, and other supervised systems, are presented and compared in chapter 2. A complete treatment of SPS and its characteristics is given in chapter 3.

1.3. Reinforcement Systems

In contrast to supervised systems, reinforcement systems require only "low quality" information from the environment. They derive the correct action for a given situation on the basis of a feedback or reinforcement signal produced by the environment during execution (figure 1.3). The feedback signal provides a rating of the quality of the solution produced by the learning system. Feedback comes infrequently, so that each step of a solution is not rated, and when it comes it only gives a gen-

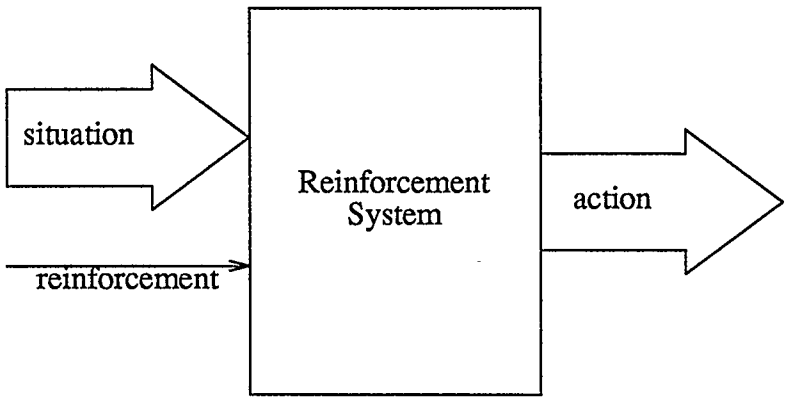


Figure 1.3. Reinforcement Learning System.

eral indication of the overall behaviour of the system. It is not detailed enough to be used directly to correct the solution. In other words, the feedback signal tells the system how good its solution is, but not why. A task is correctly solved when reinforcement is maximised. Figure 1.4 illustrates the process of feedback in a reinforcement problem. In this figure, $s-1$ situations are encountered and $s-1$ actions produced before feedback from the environment is received. In this particular case, the task is not finished, so the learning system continues to produce actions in response to situations. The next episode of reinforcement may not arrive for many more steps.

The crucial aspect of reinforcement systems is that they can derive the correct <*situation*,*action*> pairs on the basis of a signal that can be plausibly supplied by the environment. There is no longer a need for an environment which is "smarter" than the learning system. In other words, it is not necessary for some external entity to have already solved the problem, with the learning system just memorising pairs supplied from outside.

Returning to our example of the pole-cart problem, it can be formulated as a reinforcement problem by introducing a reinforcement signal which is 0 as long as the arm has not tipped past a certain angle and the cart has not reached either end of the track. When either of these is out of range, the reinforcement signal becomes -1.

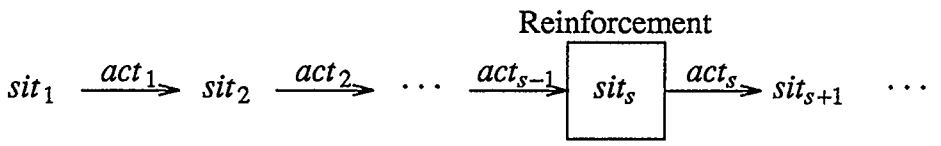


Figure 1.4. Operation of Reinforcement System.

Under this feedback scheme, the system devised by Barto, Sutton, and Anderson (1983) successfully solved the pole-cart problem.

This reinforcement signal is simple, and it is easy to see how this information could be measured and supplied to the learning system. The behaviour implied by this reinforcement signal, however, is complicated, and the difficulty lies in transforming the simple reinforcement signal into the proper, complicated, behaviour.

In order to use the indirect and infrequent reinforcement supplied by the environment, two main problems must be solved (Holland, 1986):

(1) **Generating new solutions.** Because the environment does not directly supply the learning system with correct steps in the solution, the learning system must be able to generate new solutions on its own and test their effectiveness. New solutions are generally based on old solutions, and the amount they are changed is a function of the goodness of the old solution.

(2) **Assigning credit.** Because reinforcement is infrequent, not all steps in a solution will receive direct reinforcement from the environment. Furthermore, those steps receiving direct reinforcement are receiving a rating of the effects of many actions, and therefore cannot interpret the reinforcement as a rating of only their behaviour. The system must be able to correctly evaluate the effects of a step on the overall solution.

The particular methods used by previous reinforcement learning systems to generate solutions and assign credit will be discussed in chapter 2. ESPS will be covered in chapter 4.

1.4. The Experiments

Two problems were presented to ESPS that were unsolvable by SPS. In the first, called the *single-step problem*, ESPS's task is to guess an *n*-bit number, the *target*, where the feedback is the similarity between its guess and the target. ESPS has successfully solved the single-step problem for *n* up to 128. ESPS has also been tested on an experiment where there are two target words. That is, there are two pairs, $<identifier_1, target_1>$ and $<identifier_2, target_2>$, where the goal is to produce $target_i$ when $identifier_i$ is presented. The value of *n* was 8 for this experiment. This is similar to an experiment performed in Barto, Sutton and Brouwer (1981), except that *n* was 9 for their experiment. The performance of ESPS was comparable to that of Barto, Sutton, and Brouwer's system.

The second problem, called the *multi-step problem*, is an extension of the first. In this experiment, several target words are presented, and ESPS guesses each in turn. Feedback is a single value that comes at the end of the series of guesses, and is calculated from the similarity between each guess and its corresponding target. This experiment was designed to test ESPS's ability to solve problems where reinforcement is not available after each step. ESPS has not been able to solve the multi-step problem. Discussion of this is in chapter 5.

1.5. Thesis Overview

Chapter 2 is a survey of supervised and reinforcement learning systems. Chapter 3 describes SPS in detail. It covers the architecture of SPS and the method by which it stores and retrieves data. Its unusual architecture, and its storage and retrieval methods have many implications for its behaviour, which are discussed at length. We will see that SPS is a supervised system, capable of memorisation and of solving a simple version of the best match problem. It is *not* capable of solving rein-

forcement problems. We will also see how SPS can be implemented by simple neuron-like elements. The chapter concludes with a discussion of the cerebellum, that part of the brain modelled by SPS.

Chapter 4 describes ESPS. This chapter describes the changes made to SPS, establishes that these changes give ESPS the power of a reinforcement learning system, and provides a physiological justification for the changes.

Chapter 5 describes the implementation of ESPS and the results of experiments performed with ESPS. Two experiments were run, the first being a single-step learning problem, the second a multi-step learning problem. The purpose of the first experiment is to establish that ESPS can correctly generate and test new solutions, discovering an optimal solution on the basis of indirect feedback. The second experiment extends the problem to a multi-step problem, testing ESPS's ability to assign credit correctly to steps which receive no direct reinforcement.

Chapter 6 summarises the thesis, gives conclusions, and discusses future work.

CHAPTER 2

Supervised and Reinforcement Learning Systems

This chapter reviews supervised and reinforcement learning systems and discusses their capabilities and limitations. It will be shown that supervised systems cannot solve reinforcement problems. The problems of solution generation and credit assignment are then discussed. The chapter is concluded with a review of reinforcement learning systems, discussing how each approaches the problems of solution generation and credit assignment.

2.1. Supervised Systems

This section presents short summaries of several supervised systems, all neuron-based. First discussed are two early classic models - linear associative networks (Anderson, 1983, Jordon, 1986), and the perceptron (Rosenblatt, 1962, Cohen and Feigenbaum, 1982). These represent the first attempts at neuron-based models, and were the basis for much of the work done in the 1960's on such systems. Interest waned when Minsky and Papert (1969) proved that such systems were able only to solve a limited class of problems, and the there were many interesting problems outside of this class. A simple example of such a problem is the XOR problem, discussed in section 2.1.2.

Discussed next is the Hopfield net (Hopfield, 1982), a relatively recent model which uses a quite different approach from perceptron-like models. This model has formed the basis for a new class of models, such as Boltzmann machines (Ackley, Hinton, and Sejnowski, 1985), which are more powerful than perceptrons.

This section closes with a discussion of two closely related models of the cerebellum - the Cerebellar Model Articulation Controller (Albus, 1975), and SPS (Kanerva, 1984).

It will be shown that, in addition to being able to memorise *<input,output>* pairs, each of these systems can generalise. That is, if *<input,output>* has been stored, then presentation of *input'*, a word similar to *input*, will produce *output*, or something close to it, even if *input'* has not been seen before.

Of particular interest in the discussion will be the ability of the various networks to associate arbitrary outputs with similar inputs. It will be seen that CMAC and SPS, due to a recoding of the input, have an increased ability to associate arbitrary outputs with similar inputs. In other words, they have an increased ability to discriminate among similar inputs.

2.1.1. Linear Associative Networks

A linear associative network consists of a number of components (hereafter called *neurons*), that together learn *<input,output>* pairs (Anderson, 1983, Jordon, 1986). Inputs and outputs are vectors of reals. The input vector, **x**, is distributed to each neuron. Each neuron calculates one element of the output vector, **y**. There are no connections between neurons. A linear associative network is shown in figure 2.1.

Each neuron operates in the same way, so the rest of this discussion will focus on a single neuron. A neuron with three inputs is shown in figure 2.2.

Each neuron has a vector of weights, **w**, one weight for each element in the input vector. Output y of the neuron is:

$$y = \sum_i w_i x_i = \mathbf{wx}$$

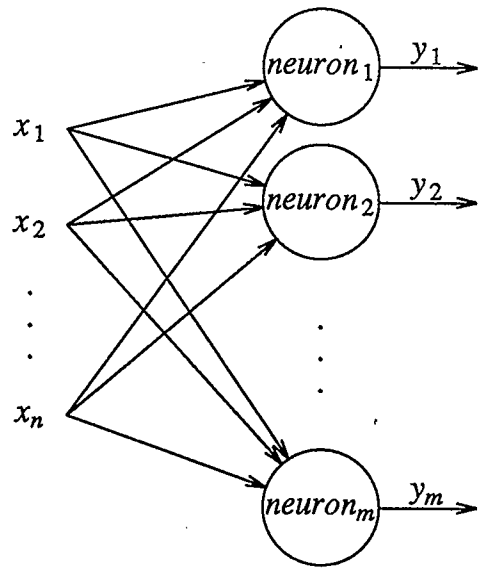


Figure 2.1. Linear Associative Network.

Weights are learned by supplying an input along with its associated output, and applying a weight update rule. There are several rules for updating weights. Shown here will be the Widrow/Hoff rule (Rumelhart, Hinton, and Williams, 1986). The Widrow/Hoff rule is

$$\Delta w_i = \eta(y_D - \mathbf{wx})x_i$$

That is, the correction applied is proportional to the difference between the desired output (y_D) and the actual output ($\mathbf{wx}$). For this reason it is also called the *delta* rule.

This rule will allow a network to learn a set of pairs $\{<\mathbf{x}_0,\mathbf{y}_0>,<\mathbf{x}_1,\mathbf{y}_1>,\ldots,<\mathbf{x}_n,\mathbf{y}_n>\}$ if all $\mathbf{x}$ are linearly independent. (A set of vectors is linearly independent if none of its elements is a linear combination of the others.)

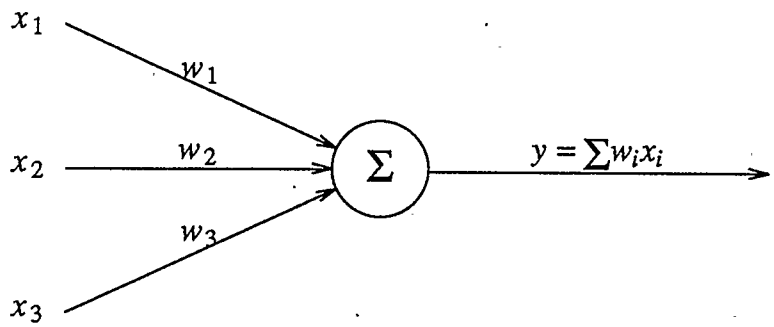


Figure 2.2. Linear Associative Network Neuron.

Linear associative networks can generalise. If <*input*,*output*> has been stored, presentation of *input'* will produce *output'*, something similar to *output*. The similarity between *output* and *output'* depends on the similarity between *input* and *input'* and the values of the weights. If *input* and *input'* are linearly separable, *output* and *output'* can be arbitrarily different. As the difference between *input* and *input'* approaches 0, so too does the difference between *output* and *output'*.

2.1.2. Perceptrons

Perceptrons (Rosenblatt, 1962, Cohen and Feigenbaum, 1982), are much like linear associative networks, consisting of a number of neurons, each of which calculates some function of an input vector. Once again, each neuron operates in the same way, so the discussion will focus on the behaviour of a single neuron. A neuron with three inputs is shown in figure 2.3. Perceptrons differ in that the inputs can only be 0 or 1, and the output is a *thresholded* linear function of input. That is, output is:

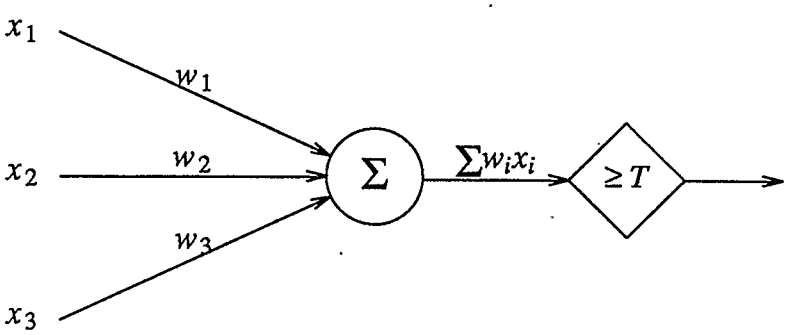


Figure 2.3. Perceptron Neuron.

$$y = \begin{cases} 1 & \text{if } \mathbf{wx} \geq T \\ 0 & \text{otherwise} \end{cases}$$

where T is a threshold, usually 0. Perceptrons are usually thought of as pattern classifiers, dividing input into two classes. Members of one class must be linearly independent of members of the other class for the perceptron to be able to classify them correctly.

Weights are updated using the Widrow/Hoff rule. That is,

$$\Delta w_i = \eta(T - \mathbf{wx})x_i$$

Note that the correction is made with respect to the unthresholded sum (**wx**), not to the actual output of the perceptron.

There is a theorem, called the *perceptron convergence theorem* (Rosenblatt, 1962), that establishes that if a set of weights exist which can classify the input set (i.e., the inputs are linearly independent), then the perceptron will converge to the correct set of weights using this learning rule.

Unfortunately, for many problems the input set is not linearly independent. For example, the XOR function cannot be performed by a perceptron as described above. It *can* be done with a two layer perceptron (see figure 2.4), but the Widrow/Hoff rule only works on single layer perceptrons.

With a two layer perceptron, the middle layer is not directly affected by either the input or the output (the perceptron in figure 2.4 is a hybrid, with two inputs to the middle layer coming directly from the input, and one coming from another neuron). Thus it is difficult to decide how middle layer weights should be changed. This is a manifestation of the credit assignment problem. This problem has been tackled by Boltzmann machines (Ackley, Hinton, and Sejnowski, 1985), and the generalised delta rule (Rumelhart, Hinton, and Williams, 1986). These systems will be discussed briefly in the section on reinforcement learning systems.

Generalisation in perceptrons differs from linear associative networks, due to the use of a threshold in the output rule. It is possible that *output* will be produced

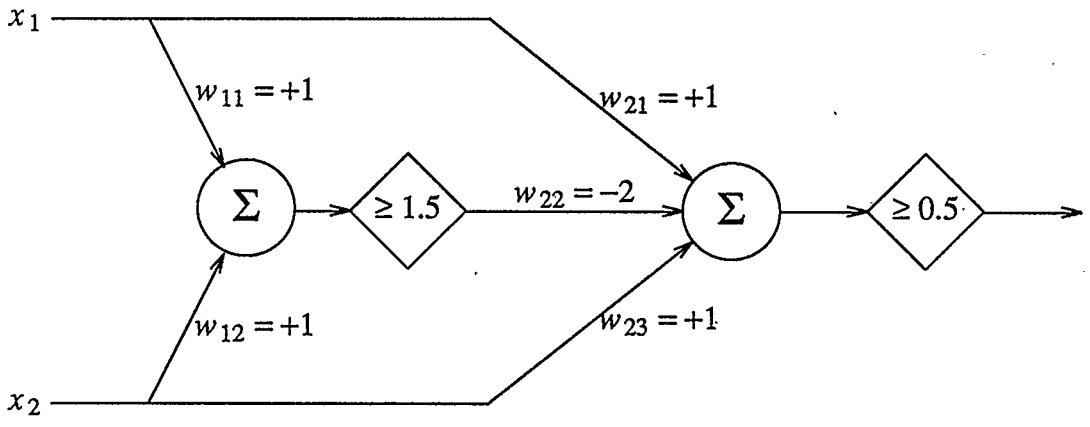


Figure 2.4. Two Layer Perceptron Solution of XOR Problem.

given *input'* (unlike a linear associative network, which would produce *output'*, different than *output*). However, once the sum produced by *input'* crosses the threshold, *input'* will not produce *output*. The change in *output* as *input'* becomes more different from *input* is a sudden one (unlike a linear associative network, where the change is gradual).

2.1.3. Hopfield Nets

A Hopfield net consists of a number of neurons with arbitrary connections between neurons (figure 2.5). Each connection has associated with it a weight, w, and each neuron has a threshold, T. Each neuron i can be in one of two states, $x_i = 0$ or $x_i = 1$. The state of the entire net is given by the vector $\mathbf{x}$ of individual neuron

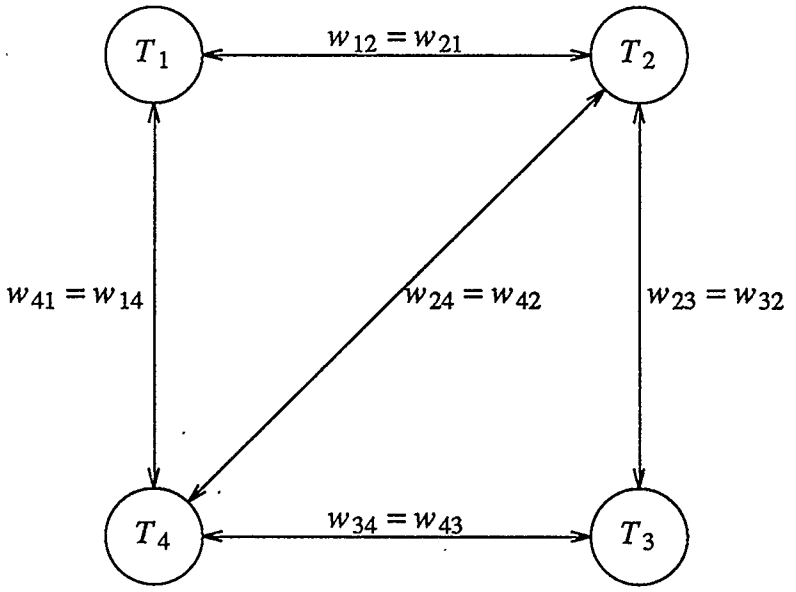


Figure 2.5. Hopfield Net.

states. Hopfield nets differ from the other supervised systems presented here in that there are no explicit input and output neurons. Some arbitrary subset of the neurons are designated as input neurons, and all the rest are designated as output neurons.

Output is generated for a given input by "clamping" the input neurons to their proper values and allowing the rest of the net to run freely, with unclamped neurons (i.e., output neurons) updating their states asynchronously, in any order, according to the following rule:

$$x_i = \begin{cases} 1 & \text{if } \sum_{j \neq i} w_{ij} x_j > T_i \\ 0 & \text{otherwise} \end{cases}$$

where T_i is the threshold of neuron i, usually assumed to be 0. Thus, neuron i sets itself to 1 if its input exceeds its threshold, otherwise it sets itself to 0.

The state of the net and the weights between neurons determine a measure of the state of the system called *energy*. Energy, E, is calculated as follows:

$$E = -\frac{1}{2} \sum_{i \neq j} \sum w_{ij} x_i x_j$$

where w_{ij} is the weight between from neuron j to neuron i. Connections are assumed to be symmetric, so that $w_{ij} = w_{ji}$. The update rule performs a gradient descent in energy space. Each update monotonically decreases E. The descent stops when a local energy minimum is reached. These local energy minima are attractors that represent outputs. When the net is started near one of these energy "wells", the update rule will cause the net to descend, and remain, in that well. The state of the output neurons when that well is reached is the output.

Energy minima are created by adjusting the weights, T, between neurons. To store a set of states, $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^n$, each weight is set using the following formula:

$$w_{ij} = \sum_s (2x_i^s - 1)(2x_j^s - 1)$$

with $w_{ii} = 0$. A given state, $\mathbf{x}^s$, represents the desired state of the *entire* network - both the state of the input neurons and of the output neurons. This rule increases the weights between two neurons i and j for a given state $\mathbf{x}^s$ if $x_i^s = x_j^s$, decreasing it when $x_i^s \neq x_j^s$. Thus, if i and j are usually on together, w_{ij} will be positive, and so i being on will increase the chances that j will be on. Similarly, if i and j usually have different values, w_{ij} will be negative, and so i being on will increase the chances that j will be off (and vice versa).

If <*input*,*output*> has been stored and the net is presented with *input'*, *output* can be produced if *input'* is close enough to *input*. If not, the net will converge on another well, and the similarity between *output'* and *output* depends on the similarities between the two wells.

2.1.4. Cerebellar Model Articulation Controller

The Cerebellar Model Articulation Controller (Albus, 1971, Albus, 1975), or CMAC for short, was developed as a model of the cerebellum. Input to CMAC is a vector of n R-ary values, $\mathbf{x}$. To facilitate the discussion of CMAC and its comparison to other systems, we assume that R is 2. Output is a vector of m reals, $\mathbf{y}$.

CMAC consists of a network of neurons, each receiving the same input, with each neuron computing one component of the output. In this sense it resembles linear associative networks and single layer networks of perceptrons. It differs in that a recoding stage is placed between the input pattern and the neurons. Also, the neurons in CMAC are hybrids of linear associative neurons and perceptrons in that inputs to CMAC neurons are binary values, as with perceptrons, while the output of a neuron is an unthresholded weighted sum of its inputs, as with linear associative neurons. CMAC is illustrated in figure 2.6. Since each neuron operates in the same way, the

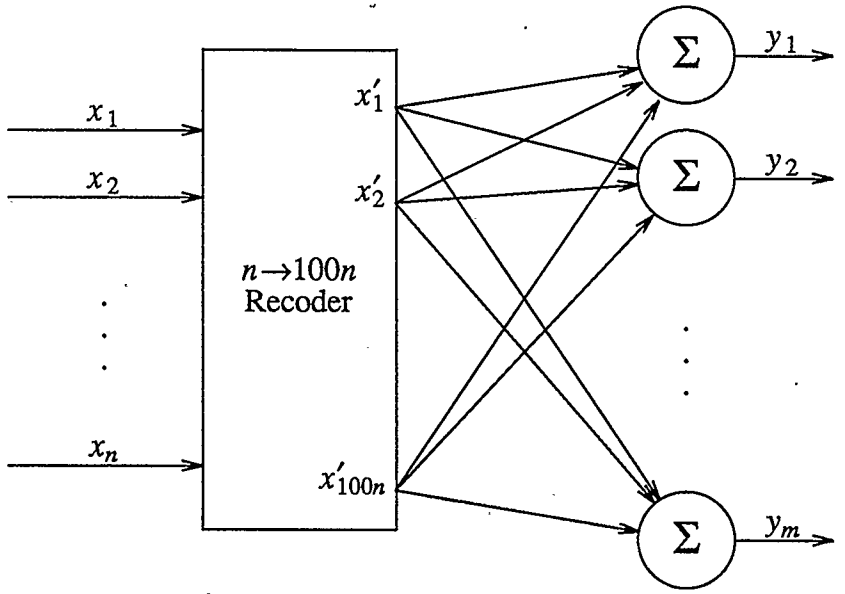


Figure 2.6. CMAC.

following discussion will deal with the properties of a single neuron. One neuron is shown in figure 2.7.

The recoding stage increases the ability of the network to produce arbitrary classifications. For example, consider the extreme case, where an n to 2^n recoder is placed between the input and the weights. For each input pattern, **x**, one line leading to a weight (x'_k) becomes 1, all others remain at 0. By setting that one weight (w_k) appropriately, the desired response can be produced. Each possible input pattern has its own weight which is adjusted independently of all others. Thus, any arbitrary classification is possible. This includes the XOR problem.

It is useful to note that a standard random access memory (RAM) in a computer has an n to 2^n recoder. An n bit address will select one of the 2^n memory locations,

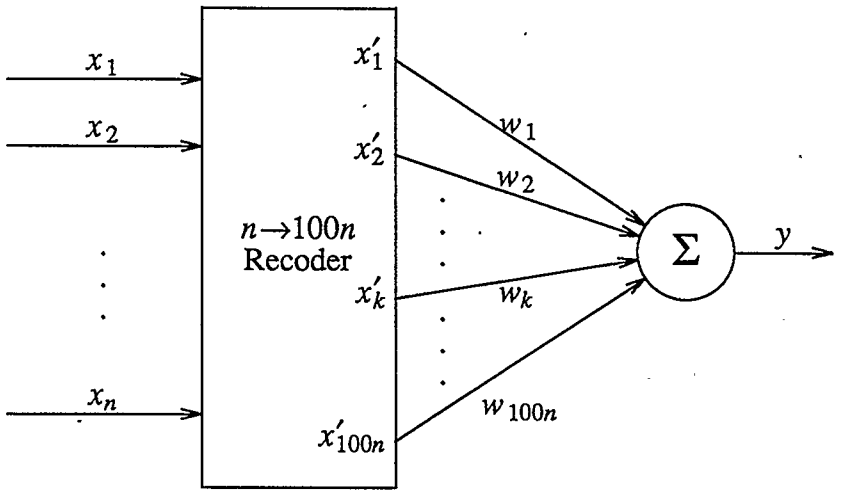


Figure 2.7. One Neuron of CMAC.

retrieving the data stored in the selected location when reading, and storing data at the selected location when writing. The address of RAM is equivalent to the input pattern, **x**, of CMAC; the data stored in RAM or retrieved from RAM is equivalent to the output, y, of CMAC. Thus, in RAM we have <*address*,*data*> pairs, as opposed to <*input*,*output*> pairs. This comparison was alluded to in section 1.2.

There are several difficulties with using n to 2^n decoder. First, for an input pattern of size n, 2^n weights are needed. For any reasonable number of inputs, the number of weights needed is impractically large. Second, no generalisation can take place between input patterns. Because the response for each input pattern is determined independently, producing the correct classification for one input does not mean that the correct classification will be produced for a similar novel input.

CMAC's approach is to use an n to $100n$ recoder. Also, instead of mapping onto one weight, each input pattern **x** is mapped onto a *subset* of approximately 1% of the $100n$ weights. The mapping is performed such that similar input patterns are mapped onto a similar subset of weights, and dissimilar input patterns are mapped onto disjoint subsets of weights. Output from CMAC is thus produced by mapping the input onto a subset of the $100n$ weights and summing those weights. That is, output is:

$$y = \sum_{i=1}^{100n} w_i x_i'$$

where $x_i' = 1$ if **x** is mapped onto x_i', and 0 otherwise.

The weight update rule is similar to the Widrow/Hoff rule. It is:

$$\Delta w_i = \eta(y_D - y)/C$$

where y_D is the desired output, y is the actual output of CMAC, and C is the number of weights which contributed to the answer. The total correction, $\eta(y_D - y)$, is thus distributed equally among contributing weights.

Generalisation in CMAC is possible because the recoding step ensures that similar inputs will be given similar encodings, thus producing similar outputs. If *input* and *input'* are sufficiently different, the encodings produced will be completely different, and so the results produced by the system can be arbitrarily different. As *input* and *input'* become more similar, the encodings will also become more similar, and so *output* will become more similar to *output'*. At some point, *output* will equal *output'*. This does not necessarily occur when *input* equals *input'*.

2.1.5. SPS

Note: the description of SPS in this section differs slightly from that given in chapter 3, the difference relating to the way data is stored in memory locations. The

variation described here was mentioned briefly in Kanerva (1984), and analysed by Chou (1987) and Keeler (1987). It is presented because it more closely resembles the other systems described in this chapter, thus facilitating its comparison to those systems.

SPS (Kanerva, 1984) is quite similar in structure to CMAC. Inputs are n bit words, outputs m bit words. Like CMAC, it performs a recoding of the input, from an n bit input onto a subset of c locations, $n \ll c \ll 2^n$. Each neuron receives the same input, and calculates one bit of the output. For this reason discussion will centre on the behaviour of a single neuron. Such a neuron is illustrated in figure 2.8.

Much like CMAC, the c locations of SPS are a subset of the possible 2^n locations determined by the input. These c locations are called *actual locations*. Each

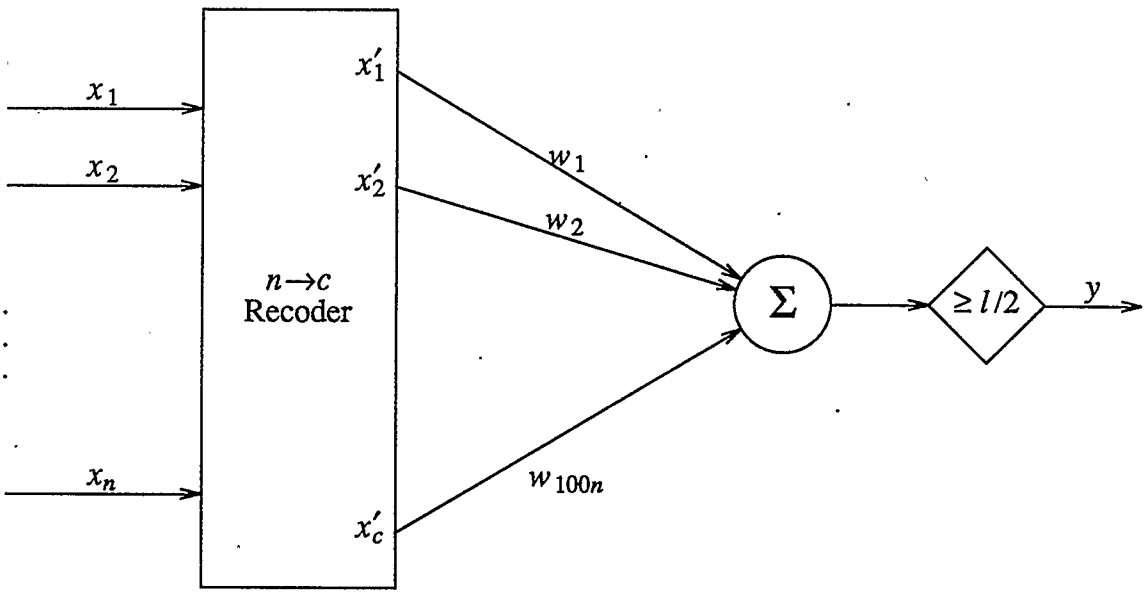


Figure 2.8. One Neuron of SPS.

actual location has associated with it an n bit identifier, chosen randomly, called its *address*.

When reading, all real locations whose address differs from the input by less than r bits (i.e., the Hamming distance between the two is less than r) respond. The weights of all responding addresses are added together to form a sum. This data is then formed into the *archetype*, which constitutes the output of SPS. The archetype is set to 0 if the bit sum is less than $l/2$, where l is the number of responding addresses, and set to 1 otherwise. Mathematically,

$$y = \begin{cases} 1 & \text{if } \sum_{i=0}^{c} w_i x'_i \geq l/2 \\ 0 & \text{otherwise} \end{cases}$$

where $x'_i = 1$ if $\mathbf{x}$ maps onto x'_i, and 0 otherwise.

When writing, all real locations within r bits of the input take part in writing. For responding address k, w_k is incremented by 1 if the desired output is 1, otherwise it is decremented. That is,

$$\Delta w_k = \begin{cases} 1 & \text{if } y_D = 1 \\ -1 & \text{otherwise} \end{cases}$$

Unlike CMAC, weights are updated by a fixed amount (1 or −1), not an amount proportional to the difference between the desired and actual output.

The mechanism for generalisation in SPS is similar to CMAC. Generalisation occurs because the recoding step ensures that similar inputs will be given similar encodings, thus producing similar outputs.

2.1.6. Summary

Supervised systems can memorise *<input,output>* pairs. Of particular interest here is SPS. The recoding stage of SPS (and CMAC) increases its ability to memor-

ise arbitrary classifications, compared to other related systems. Like the other supervised systems reviewed, SPS has the ability to generalise. Finally, it is proposed as a model of the cerebellum, which makes SPS interesting not only as a learning system, but also as a system which can be used to study the brain.

As a supervised system, though, SPS is limited as to the problems it can solve. It requires an environment which possesses as much knowledge about the task as SPS is expected to learn. In other words, the environment must be able to solve the problem already. Reinforcement systems do not suffer from this limitation. They can deal with an environment which does not possess (explicit) knowledge about the task to be learned. All that is required is a rating of the solution produced by the learning system. In many problems, such as the pole-cart problem, this rating is easily and plausibly supplied.

From the standpoint of the human brain learning physical tasks, it seems that the brain must be able to do reinforcement learning. First, learning of some sort takes place because performance improves with practise. If no learning took place, performance levels would remain constant. Second, the learning is reinforcement learning, since the brain is only given general feedback on the quality of its performance. It is not told by some external source which muscle movements were incorrect, and, more importantly, what the correct movements are.

Therefore, we can conclude that the brain does reinforcement learning. This does not necessarily imply that the cerebellum, that part of the brain modelled by SPS, is the place where reinforcement learning takes place. Nevertheless, if a cerebellar model such as SPS could be made into a reinforcement learning system, and if this could be done in a physiologically plausible way, this would suggest that the cerebellum is capable of reinforcement learning.

Before describing ESPS, an extension of SPS which solves reinforcement learning problems, we first look at previous approaches to reinforcement learning.

2.2. Reinforcement Learning Systems

This section begins by discussing the problems of credit assignment and solution generation, two problems that a reinforcement learning system must overcome. Several reinforcement learning systems and their approaches to these two problems are then discussed. The systems discussed are: Holland's classifier system (Holland, 1986), simulated annealing (Kirkpatrick, Gelatt, and Vecchi, 1983), stochastic learning automata (Narendra and Thathachar, 1974), adaptive threshold systems (Widrow, Gupta, and Maitra, 1973), and the associative search network (Barto, Sutton, and Brouwer, 1981).

Before discussing these systems, a word should be mentioned about three important systems which will not be discussed in detail: Samuel's checker player (Samuel, 1963, Cohen and Feigenbaum, 1982), Boltzmann machines (Ackley, Hinton, and Sejnowski, 1985), and the generalised delta rule (Rumelhart, Hinton, and Williams, 1986).

Samuel's checker player is a classic early attempt at machine learning. The goal was to develop a program that learned to play checkers based on playing experience. Samuel had to deal with the problems of credit assignment and generating new solutions. However, his approach included creating a "smart" environment which would supply the checker player with good moves. The checker player would then adjust its performance on the basis of the difference between its move and the supplied move.

The supplied moves came in the form of book moves (moves taken from a book of play between two master checker players) or a "performance standard"- a move generated by doing a deeper look-ahead search.

The key is that the environment supplies the checker player with "high quality" information. Samuel's checker player cannot operate in the environment shown in figure 1.3 and so it is not included in the survey of reinforcement learning systems.

The other two systems, Boltzmann machines and the generalised delta rule, are two neuron-based systems which have been developed recently. These systems were developed to address the problem of properly adjusting weights in multi-layer neural networks.

Recall that the Widrow/Hoff rule could only find the proper weights in single layer networks. This task was easily solved because each weight is directly determined by the desired input and output. In a two-layer network, this is not the case, as there are weights not directly connected to the input or the output. The effectiveness of a choice of values for these weights can be judged only by the effect they have on the overall behaviour of the network. Thus this is an instance of the credit assignment problem. Credit must be properly assigned to segments of a "solution", where the solution is the set of weights in the network.

Both Boltzmann machines and the generalised delta rule (also called *back propagation*) have developed a weight update rule which can be successfully applied to multi-layer networks. Both work to optimise an internally generated error signal measuring the difference between their behaviour and the desired behaviour as given by the environment. Like Samuel's checker player, however, they are not included in the review of reinforcement learning systems because they cannot operate in the environment shown in figure 1.3. Both require explicit *<situation,action>* pairs, not a simple reinforcement signal.

The following two sections define the problems of credit assignment and solution generation. This will be followed by a review of several reinforcement systems,

and concluded by a summary of their approaches to credit assignment and solution generation.

2.2.1. Credit Assignment

The credit assignment problem appears in various guises. A classic example of the problem is in the game of chess. When a move in chess leads directly to a win, that move is good. This is less true of earlier moves. Did those moves contribute to the winning situation at the end of the game, or was the game won in spite of those earlier moves? This is a *temporal* form of the problem. Credit has to be assigned to different steps over time.

The credit assignment problem also occurs in a *structural* form. If the results of various components of a system are combined together to form a solution, failure or success of that solution can depend on any or all of the components. Discovering which components are important and which are not is a credit assignment problem. Those components which play a large role in the solution should be given corresponding weight. Boltzmann machines and the generalised delta rule both deal with structural credit assignment.

2.2.2. Solution Generation

Figure 1.3 makes it clear that a reinforcement learning system is not supplied with the action to be performed in a situation. It must therefore be able to generate its own solutions.

The size of the solution space makes a random search for new solutions impractical. Instead, reinforcement systems create new solutions based on existing solutions. How this is done varies considerably.

Some systems maintain, for each situation, one action. In some systems, if that action leads to success, it is reinforced - it will take more negative reinforcement in the future to change the action. Others make small changes to the action, measure its effect, and accept the change if it improves the solution or satisfies some either criterion. This means that ratings must be maintained on the goodness of the old solution. These ratings are produced via credit assignment.

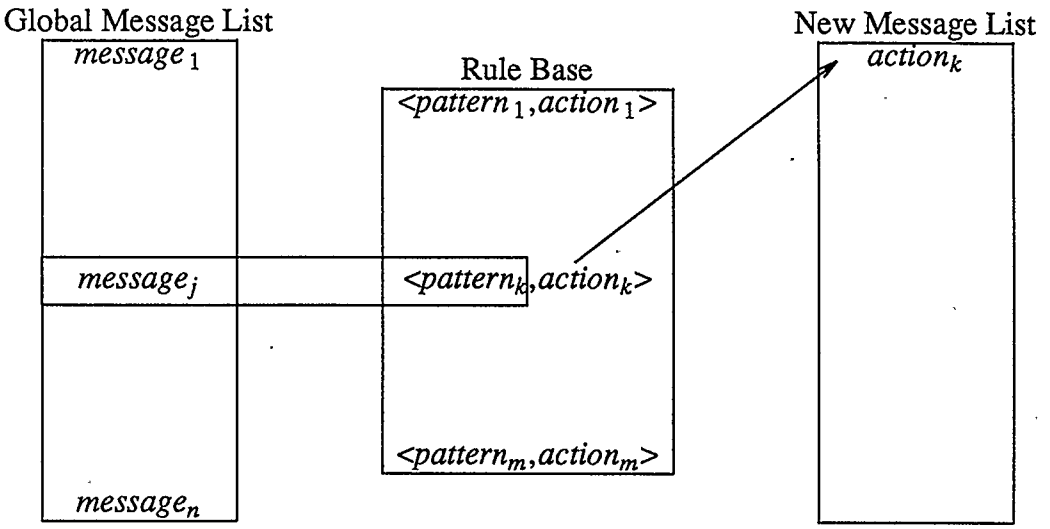
Other systems maintain a probability distribution for each situation. One of a number of actions is possible for that situation, with a probability of being selected determined by the distribution. Changing a solution means altering the distribution. Most systems make the change on the basis of the difference between expected and actual reinforcement, which means that, as above, ratings must be maintained.

The choice of representation and update policy determines what kinds of problems are solvable. For example, a system which maintains one action per situation and updates that action by testing small changes and accepting any that improve its performance is performing a gradient descent in action space. Such a system must deal with the problem of getting trapped in local minima.

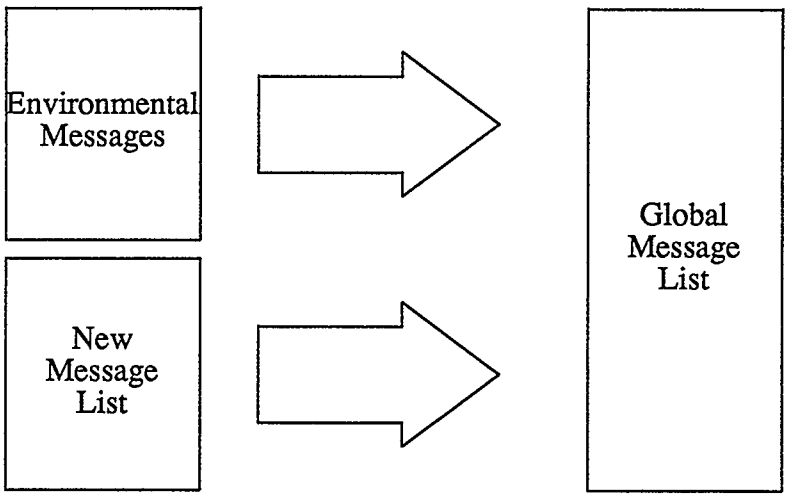
2.2.3. Holland's Classifier System

Holland's classifier system (Holland, 1986) is a rule-based system, where rules are in the form of *<pattern,action>* pairs called *classifiers*. All activity takes place on a *global message list*, a set of messages originating from the environment and from classifiers. The *pattern* portion of a classifier is a template which can match any one of a set of messages. If a matching message is found on the global message list, then the *action* portion of the classifier, which is itself a message, is added to a new global message list (subject to restrictions mentioned in the next paragraph). This process is shown in figure 2.9(a), where $pattern_k$ of classifier k matches $message_j$, causing

$action_k$ to be placed on the new global message list. Adding an *action* to the new



(a) Matching Procedure



(b) Constructing New Global Message List

Figure 2.9. Holland's Classifier System.

message list can have effects on the environment. After all classifiers have been checked against all messages on the global message list, the old list is removed, and a new one is created from the new global message list and any new messages originating from the environment (figure 2.9(b)). This process is then repeated.

Each classifier has a rating (called "strength" by Holland). In order to place a message on the message list, a classifier has to bid for the privilege - the highest bidders get their messages placed on the message list. A winning bidder has the amount of its bid subtracted from its strength. The bid strength of a matching classifier C at time t is

$$Bid(C,t) = cR(C)Strength(C,t)$$

where c is a constant less than 1 (e.g., 1/16), and $R(C)$ is a measure of the specificity of C - the more specific C is, the greater the value of $R(C)$ (one classifier is more specific than another if the set of messages with which it matches is smaller than the other's). The strength of C becomes

$$Strength(C,t+1) = Strength(C,t) - Bid(C,t)$$

A classifier can regain strength if its message is matched by later classifiers who successfully bid. The amount of strength it regains depends on the bid strengths of the later classifiers and the number of other messages matched by those later classifiers. That is, if k messages are matched by a later classifier C', and C's message is among those k messages, then

$$Strength(C,t+2) = Strength(C,t+1) + Bid(C',t+1)/k$$

Classifiers that lead directly to environmental reinforcement receive that reinforcement.

Thus, Holland's classifier system deals with credit assignment through the use of strengths and the process of bidding. Those classifiers which are useful will gain

strength, either directly by receiving reinforcement from the environment, or indirectly by having its message matched by other useful classifiers.

New classifiers are generated using the genetic algorithm. A subset of existing classifiers are chosen according to their strengths - the higher the strength, the more likely the classifier will be chosen. The classifiers in the subset are paired off, and genetic operators are applied to the pairs, producing new classifiers.

The newly produced classifiers replace the weakest classifiers. If these new classifiers are useful, they will gain strength and so will likely not be replaced in the future. If they are not useful, they will not gain strength, and so will likely be replaced in the future.

So, the genetic algorithm produces new classifiers on the basis of the best old classifiers. These new classifiers replace the weakest old classifiers. The changes produced are based on a probabilistic procedure, both in how the candidate classifiers are chosen, and how they are combined.

2.2.4. Simulated Annealing

Simulated Annealing (Kirkpatrick, Gelatt, and Vecchi, 1983) is a method for solving problems of combinatorial optimisation. The goal in combinatorial optimisation is to find the minimum of a function of the configuration of a system of very many independent components. A configuration can be viewed as a solution to a problem.

The function to be minimised (called the *cost* function) measures the "goodness" of the given configuration - the lower the cost, the better the solution. Total distance in the travelling salesman problem is an example of such a function.

If we view the cost function as a reinforcement signal from the environment, we see that minimising cost is equivalent to maximising reinforcement. Since the cost function is simple to calculate and thus plausibly supplied by the environment, this qualifies as a reinforcement learning problem.

Simulated annealing is based on the ideas of statistical thermodynamics (Nash, 1974), and so much of the terminology is borrowed from that discipline. Ergo, we speak not of "cost" but of "energy". A given configuration has an energy, E, given by the cost function. The goal of simulated annealing is to find the minimum energy configuration. The system starts with an arbitrarily picked configuration, which will have some energy. The configuration is given a small random change, which will change the energy of the configuration by ΔE. This change is accepted immediately if the change lowers the energy of the configuration. If the change raises the energy, then it is accepted with probability

$$P(\Delta E) = e^{\frac{-\Delta E}{k_b T}}$$

where k_b is Boltzmann's constant. This is the process by which new solutions are generated. One can view simulated annealing as a hill climbing search with noise, where the amount of noise is determined by T.

The T parameter represents the temperature of the system, another concept borrowed from statistical thermodynamics. The system is started out at a high temperature, which means that nearly all changes will be accepted. This effectively randomises the system. The temperature is then lowered according to a fixed schedule. As T is lowered, statistical thermodynamics tells us that the probability of being in a low energy state increases. The reason for not starting with T low is that at low temperatures the time to make transitions out of sub-optimal energy minima is high. The hope is that by starting at a high temperature and slowly lowering it, T will pass

through a value for which transitions to the global minimum are easy enough for it to occur within a reasonable time, but for which transitions out of the global minimum are difficult enough that the system stays in the global minimum.

The process of slowly lowering T in a physical system is called *annealing*. Thus this process of lowering the T parameter is called *simulated* annealing.

Simulated annealing does not deal with temporal problems, except those which have a fixed time span, such as the variation of the travelling salesman problem tackled in Kirkpatrick, Gelatt, and Vecchi (1983). In this variation, a solution always requires c steps, where c is the number of cities. The entire solution can be represented by a single configuration. The credit assignment problem is thus structural. The simulated annealing system maintains a record of the overall goodness of a solution (via E), but does not try to assign credit to individual parts of a solution. However, when changes to a part of the solution increase E (i.e., result in a poorer solution), the probability of their acceptance decreases with the increase in E. It is through this mechanism that properly selected portions of a solution are maintained.

2.2.5. Stochastic Learning Automata

Structurally, stochastic learning automata are very simple. Input from the environment consists of a single reinforcement signal, with 0 signifying positive reinforcement, and 1 negative reinforcement. Since the automaton has no input other than the reinforcement signal, there is no concept of a current situation. To solve a problem which requires different responses in different situations would require one automata per situation.

Output consists of one of p actions, $\alpha_1, \ldots, \alpha_p$. The automaton maintains an internal probability vector $\mathbf{p}$, with one probability per action, which governs the choice of the next action. That is, action α_k is selected with probability p_k. The en-

vironment is random. That is, for a given output, α_k, it produces a penalty with probability c_k, where **c** is a vector of penalty probabilities. The vector **c** has one probability per input action. Note that the environment's response is dependent only on the current action - it does not depend on previous actions. Environments can be *stationary*, in which case c does not change over time, or *nonstationary*, in which case c does change. For this discussion, a stationary environment is assumed.

The goal is to modify **p** such that reinforcement is maximised. A stochastic learning automata and its environment are illustrated in figure 2.10.

Many learning schemes have been tried. All are based on the idea that when action α_i is chosen at time t and positive reinforcement is received, then $p_i(t)$ (i.e., p_i at time t) should be increased and all other components of p decreased. Similarly, when negative reinforcement is received, then $p_i(t)$ should be decreased and all others in-

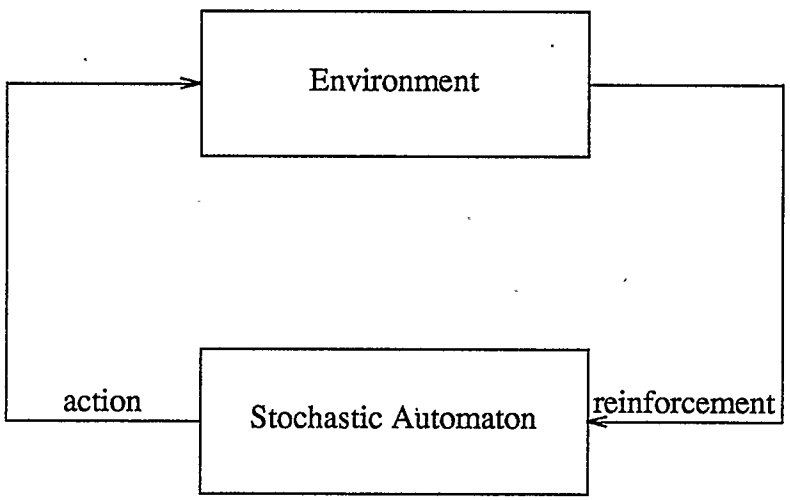


Figure 2.10. Stochastic Learning Automaton.

creased. Some schemes, such as the one discussed below, do not update **p** under certain conditions.

The vector **p** represents the learning automaton's current solution to the problem. By adjusting individual action probabilities, the solution is changed. This is how new solutions are generated. The same mechanism deals with the credit assignment problem - no ratings are kept of solutions; rather, ratings are maintained indirectly, by increasing the probabilities of actions which lead to reward and decreasing all others. Note that the credit assignment problem here is structural, as reinforcement is received after every step.

The simplest learning scheme is called linear reward-inaction. When action α_i is performed and positive reinforcement is received, then all other probabilities are updated by

$$p_j(t+1) = p_j(t) - ap_j(t)$$

where $0 < a < 1$. The action probability for action i is updated by

$$p_i(t+1) = p_i(t) + \sum_{j \neq i} ap_j(t).$$

The "inaction" part of the name comes into play when negative reinforcement is received. In this case, nothing is done to p.

The performance of this scheme is ε-*optimal*. To understance what ε-optimality is, optimality should first be defined. An optimal scheme is one which, as time goes to infinity, chooses action c_l with probability 1, where action c_l produces the greatest positive reinforcement. An ε-optimal scheme can be made as close to optimal as desired by proper choice of parameters. That is, the expected value of reinforcement can be made arbitrarily close to c_l.

2.2.6. Adaptive Threshold Systems

Adaptive threshold systems (Widrow, Gupta, and Maitra, 1973) are based on linear threshold systems, namely perceptrons. Unlike perceptrons, adaptive threshold systems do not require <*input*,*output*> pairs to be explicitly supplied.

Feedback consists of a simple on/off signal - the system can receive either reward or punishment. When it receives reward it updates its weights so as to decrease the difference between the output of the system (y) and the unthresholded output ($\mathbf{wx}$). The Widrow/Hoff rule is used, giving:

$$\Delta w_i = \eta(y - \mathbf{wx})x_i$$

This increases the chances that y will be produced in the future, given the input $\mathbf{x}$. That is, changes in the weights produced by other updates will be less likely to change the output of the system when $\mathbf{x}$ is encountered in the future. In this sense, the pair $\langle \mathbf{x}, y \rangle$ has a rating associated with it. The more extreme the output, the higher the rating. An output near the threshold indicates a low rating.

When it receives punishment it updates its weights so as to decrease the difference between the inverse of the output of the system (i.e., $1-y$) and the unthresholded output. Once again, the Widrow/Hoff rule is used, but the desired output is the opposite of what was actually produced. That is:

$$\Delta w_i = \eta((1-y) - \mathbf{wx})x_i$$

Since outputs are either 1 or 0, the desired output will be $1-y$, where y is the actual output.

This decreases the chances that y will be produced in the future, given $\mathbf{x}$. If the unthresholded output was near the threshold, it is possible that the weight change will place $\mathbf{wx}$ on the other side of the threshold, and thus change y. This is the mechanism by which new solutions are generated. If the output is incorrect, a pair will receive

negative reinforcement. If this process continues long enough, the output, y, of the pair will change value. Pairs which begin with the output near the threshold will require less negative reinforcement to change output than pairs which have an output far from the threshold.

Because it is based on perceptrons, an adaptive threshold system cannot learn anything that a perceptron cannot. Therefore, inputs must form a linearly independent set.

In Widrow, Gupta and Maitra (1973), the adaptive threshold system is applied to the game of blackjack. Input to the system is the current situation, which is composed of the value of the upturned dealer's card, the current total of the system's hand, and whether aces are high or low. The action produced in a given situation is whether to draw another card ("hit") or stay with the current cards ("stick"). The decision to stick is final.. One cannot draw another card after deciding to stick.

Learning proceeds as follows. One hand is played, which the player either wins or loses. This is done without any reinforcement taking place. The hand is then replayed exactly, except this time the reinforcement is set appropriately (reward for winning the hand, punishment for losing the hand) and weights are updated for each <*situation,action*> pair encountered in playing that hand. Because reinforcement is determined by the outcome of the hand, rather than the merit of each decision, some incorrect decisions will be rewarded (if the hand is won in spite of those decisions), and some correct decisions will be punished (if the hand is lost in spite of those decisions). Thus, while reinforcement is received on each step, it is not reliable.

The system asymptotically approached the optimal strategy (called the Thorp optimal strategy). Under the Thorp optimal strategy, a player can expect to win 49.5% of all games. Results of the adaptive threshold system varied, depending on

the value chosen for the learning parameter, η, with the best result being about a 48% winning rate (achieved after approximately 10,000 games). The initial winning rate, before any learning had been done, was about 22%.

As mentioned before, because it is based on perceptrons, the adaptive threshold element can only properly classify linearly separable sets. Thus, the key to the success of this system in the domain of blackjack was an encoding scheme that translated inputs into a linearly independent set.

2.2.7. Associative Search Network

The associative search network (Barto, Sutton, and Brouwer, 1981), or ASN for short, is a system based on single-layer perceptron networks. Input to the system consists of a vector **x** of reals, output is a vector **y** of 1's and 0's. ASN also receives a feedback signal, *z*, from the environment. An ASN and its environment are shown in figure 2.11.

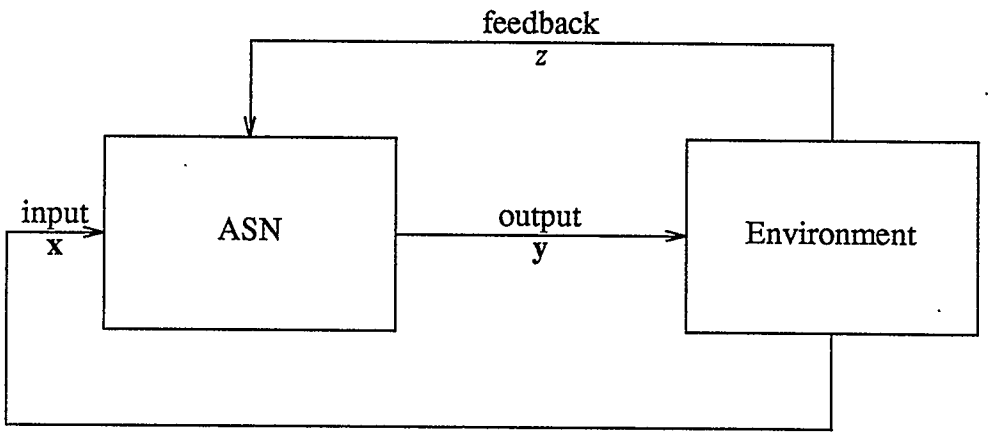


Figure 2.11. Associative Search Network and Environment.

If **y** has m elements, then ASN will have m+1 neurons; the first m neurons produce **y**, and the m+1st neuron produces a prediction of reinforcement used by the other m neurons. An ASN is shown in figure 2.12. Each neuron (with the exception of the predictor) operates similarly, so the following discussion will focus on a single neuron.

More specifically, output y from a neuron at time t is

$$y(t) = \begin{cases} 1 & \text{if } \mathbf{w}(t)\mathbf{x}(t)+NOISE > 0 \\ 0 & \text{otherwise} \end{cases}$$

where *NOISE* is a normally distributed random variable. This output function is identical to a perceptron's except for the inclusion of noise. The addition of noise means that the weighted sum **wx** determines the *probability* of an output, rather than the output itself. This means that at any time, any output is possible, with the probability of a given output being dependent on the weighted sum - the more extreme the value of the sum, the less likely it is that this neuron will give a result different from that of a regular perceptron. This is the mechanism by which new solutions are generated.

These neurons also differ from regular perceptrons in that they receive reinforcement from the environment, in the form of a single real value, z. The reinforcement is used in the weight update rule, which is

$$w_i(t+1) = w_i(t)+\eta[z(t)-p(t-1)][y(t-1)-y(t-2)]x_i(t-1)$$

Note that the reinforcement, z, for an action at time $t-1$ comes at time t. The value $p(t-1)$ is the output from the predictor neuron, and represents its prediction of $z(t)$.

The change in w_i then, depends on three things: the difference between expected and actual reinforcement ($z(t)-p(t-1)$), the change in output ($y(t-1)-y(t-2)$), and the actual input ($x_i(t-1)$). When any of these is 0, so too is the change in w_i.

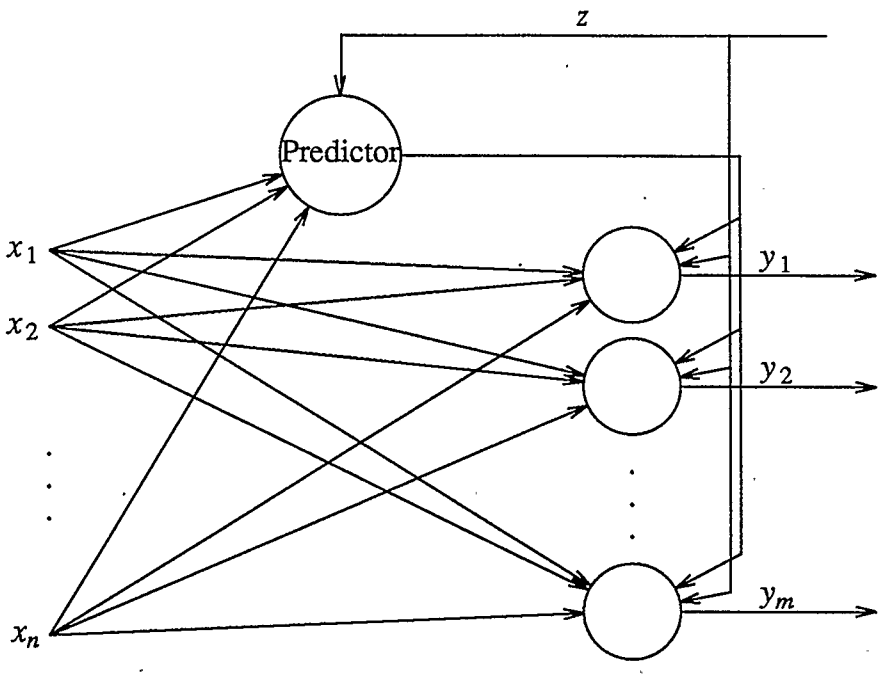


Figure 2.12. Associative Search Network.

The reinforcement predictor neuron has its own set of weights, output rule, and weight update rule. The output of the neuron is

$$p(t) = \mathbf{w}^p \mathbf{x}$$

where $\mathbf{w}^p$ is the vector of weights for the predictor. The weight update rule is

$$w_i^p(t+1) = w_i^p(t) + \eta^p [z(t) - p(t-1)] x_i(t-1)$$

where η^p determines the rate of learning for the predictor. This is the familiar Widrow/Hoff rule, where $z(t)$ is the desired output and $p(t-1)$ is the actual output.

ASN does not deal with multi-step problems, so only structural credit assignment is dealt with. Credit assignment is accomplished through the combination of expectation, as maintained by the predictor neuron, and the strengths of the weights

of the output neurons. The expectations are used to decide whether to reward or punish a neuron, and the strengths of the weights of the output neurons are adjusted to reflect the reward or punishment. Rewarded neurons will have their weights strengthened, and punished neurons will have their weights weakened.

While ASN doesn't deal with a temporal form of credit assignment, an extension of ASN called ASE/ACE (Barto, Sutton, and Anderson, 1983), does. ASE/ACE (**A**ssociative **S**earch **E**lement/**A**daptive **C**ritic **E**lement) resembles ASN, with an expanded role for the predictor neuron. Essential to ASE/ACE's ability to deal with temporal credit assignment is the use of expectations of reinforcement of subsequent steps in determining the reinforcement given to the output neurons for the current step. By taking into account future expectations, it can assign credit to steps which receive no credit directly. This is closely related to the scheme used by Holland's classifier, in which classifiers are strengthened if they lead to environmental reinforcement or if they lead to other useful classifiers.

2.2.8. Summary

To be a reinforcement system, we have seen that a system must be able to deal with the problems of credit assignment and generating solutions. Dealing with the credit assignment problem means maintaining a rating system, either implicitly or explicitly. It is interesting to note that the only two systems reviewed which handle the temporal credit assignment problem (i.e., Holland's classifier system and ASE/ACE) are the only ones to maintain an explicit rating system.

The rating system plays a part in the generation of new solutions. In the systems reviewed, new solutions are generated on the basis of old solutions, the variation in the new solution being dependent on the rating of the old solution.

The next chapter describes SPS in detail. This is a prelude to chapter 4, which describes ESPS, an extension of ESPS that can solve reinforcement learning problems.

CHAPTER 3

Self-Propagating Search

This chapter describes Pentti Kanerva's (Kanerva, 1984) Self-Propagating Search (SPS). The architecture of SPS is given, and its method of reading and writing data is described. The properties which come about because of its architecture are described. In particular, it is shown how SPS can solve a limited version of the best match problem. Finally, the cerebellum and two theories thereof which are related to SPS are discussed.

In addition to Kanerva (1984), SPS has been described and analysed in Chou (1987) and Keeler (1987). Specifically, Chou and Keeler analyse the memory capacity of SPS (which they call Sparse Distributed Memory, or SDM). To my knowledge, SPS has not been applied to any problems.

3.1. Description of SPS

SPS is a supervised learning system based on neuron-like components, designed to solve the best match problem. It was described briefly in chapter 2.

Addresses in SPS are n-bit binary numbers, where n is typically large (>100). Data (also called *words*) are m-bit binary numbers. For the remainder of this chapter and the next, assume for simplicity that $n = m$.

SPS is based on the properties of n-dimensional binary metric spaces, where the distance metric is Hamming distance. The Hamming distance between two words is the number of bits at which the two words differ. Therefore, the Hamming distance

between 100 and 100 is 0*, the distance between 100 and 101 is 1, and the distance between 100 and 011 is 3.

In a normal random-access memory with n-bit addresses, there are 2^n locations in memory, one for each possible address. The word "location" refers to a physical location in memory. Locations have a label, called their *address*. In a normal random-access memory, locations are selected by broadcasting an address. The location whose address matches the broadcast address responds.

For the size of n dealt with by SPS, having 2^n actual addresses is impractical. Therefore, in SPS only a small subset of all potential 2^n locations actually exist. These locations are called *actual* locations. Since only a small subset of all potential locations exist, each actual location serves many addresses. The addresses it serves are determined by the address of the actual location (these are randomly assigned), and by a memory-wide constant called the *read/write circle size* (r for short). An actual location responds to all addresses within r bits of its address. A further difference between SPS and standard random-access memories is that in SPS, several data words are stored at each actual location (standard random-access memories store only one).

Figure 3.1 shows a memory in which $n = 4$ (i.e., there are $2^n = 16$ total addresses), $r = 1$, where there are 5 actual locations, and where each location stores 2 words. When an address is presented, each actual location checks if the presented address is within r bits of its own address. If it is, then it responds. For example, when the address 0000 is presented, two actual locations respond: 0000, which is 0 bits away, and 1000, which is 1 bit away. The locations 0101, 1011, and 0110 do not respond because they are all more than r bits away.

*Equivalent phrases are "the distance between them is 0 bits" and "they are 0 bits apart".

actual location	data
0000	0011
	0001
0101	1101
	0010
1011	0111
	1110
0110	0010
	0101
1000	1110
	0000

$r = 1$

Figure 3.1. Example Memory.

3.1.1. Writing

To write a word w at an address a means adding w to the list of words at each of the actual locations within r bits of a. When writing, the oldest word in the list is replaced by the word being written.

Figure 3.2 shows the operation of writing a word, $w = 1001$, at an address, $a = 0001$. For figure 3.2 we assume that we start with the same situation as in figure 3.1. When w is written at a, two addresses respond (0000 and 0101 - these are highlighted in the diagram). Therefore w is added to the head of the list at each of these locations (we assume that the head of the list is at the top). The oldest member of each list is lost. This means that at location 0000, 0001 is lost, and at location 0101, 0010 is lost. The state of the memory after writing is shown in figure 3.2.

actual location	data
0000	**1001**
	0011
0101	**1001**
	1101
1011	0111
	1110
0110	0010
	0101
1000	1110
	0000

$r = 1$

Figure 3.2. Writing to Memory.

3.1.2. Reading

To read at an address a, we gather together the lists at the responding actual locations and form what is called the *archetype*. The archetype is a "representative" of the words in the list, and is formed by a bitwise application of the majority rule. That is, if there are more 1's than 0's in a given bit location, the archetype has a 1 at that location. If 0's are more populous, the archetype has a 0. Another way to look at it is by considering bit sums. The bits at each bit position are summed. If, in a given position, the sum is greater than half the number of summands, then the archetype has a 1 at that position, otherwise it has a 0.

Figure 3.3 shows a read at $a = 0001$, the same location at which w was written. Once again, $r = 1$. While we shouldn't necessarily expect to read back w, we should read back something quite similar to w. More precisely, what we read back now should not be farther from w than if we had read before writing w.

actual location	data
0000	**1001**
	0011
0101	**1001**
	1101
1011	0111
	1110
0110	0010
	0101
1000	1110
	0000

$r = 1$

(a) Memory.

	b_3	b_2	b_1	b_0	
pooled data	1	0	0	1	
	0	0	1	1	
	1	0	0	1	
	1	1	0	1	
bit sums	3	1	1	4	1's
	1	3	3	0	0's
archetype	**1**	**0**	**0**	**1**	

(b) Formation of Archetype.

Figure 3.3. Reading from Memory.

In any case, the address $a = 0001$ once again cause 0000 and 0101 to respond. The lists from each of them are gathered together and the archetype is formed (figure 3.3(b)). As an example, let's detail the calculation of the left-most bit of the archetype (b_3). Counting the number of 1's and 0's in b_3 of the accumulated words gives us a total of three 1's and one 0. Since 1's are more populous than 0's, b_3 of the archetype is 1. The same procedure is performed for b_2, b_1, and b_0, giving us an archetype of 1001, which was what was written previously.

3.2. Properties of SPS

In each of the following examples of properties of SPS, assume that we start with the memory in figure 3.2 and figure 3.3(a).

3.2.1. Reading at a Similar Address

Suppose that a word, w, has been written at an address, a, and that reading at a produces w. Reading at an address, a', very similar to a will produce a word very similar to w, since most of the locations responding to a' will be the same ones which respond to a. This property is useful when dealing with noisy input data (i.e. corrupted addresses). It is also useful for handling novel situations, if those novel situations are similar to ones stored before, and if it is acceptable to perform similar actions in similar situations.

The effect of reading at an address close to $a = 0001$, which we'll call a', and whose value is 0000, is shown in figure 3.4.

Reading at a' causes two locations to respond. One of these locations (0000) also responds to a. The archetype is 1011 (the calculation of the archetype is shown in figure 3.4(b)), which is similar to what is produced (1001) when reading at a. (Note that when the number of 1's equal the number of 0's in a bit position, the archetype is 1). Thus, by moving away from the original write address, a, the original data is no longer recovered, but something close to it is. As a' becomes more different from a, the less likely it is that the original data is recovered. The reason for this is that, as a' moves away from a, fewer responding locations contain copies of the original data. The fewer copies there are, the less they influence the archetype, and the less likely it is that the archetype will resemble them.

actual location	data
0000	**1001**
	0011
0101	1001
	1101
1011	0111
	1110
0110	0010
	0101
1000	**1110**
	0000

$r = 1$

(a) Reading at $a' = 0000$.

	b_3	b_2	b_1	b_0	
pooled data	1	0	0	1	
	0	0	1	1	
	1	1	1	0	
	0	0	0	0	
bit sums	2	1	2	2	1's
	2	3	2	2	0's
archetype	**1**	**0**	**1**	**1**	

(b) Formation of Archetype

Figure 3.4. Reading at a Similar Address.

Now, further assume that another word, w_1, has been written at an address, a_1, which is similar to a. Then, if a' moves from a to a_1, the archetype will gradually change from w to w_1, as the influence of w in the formation of the archetype wanes and the influence of w_1 waxes.

From a practical standpoint, this is useful for "filling in" missing parts of the task being learned. If SPS encounters a situation similar to other situations at which actions have been stored, the action returned will be a kind of average of the actions stored with those various situations. Note that this averaging is not explicitly performed. Rather, it is a natural emergent property of the storage and recall scheme of

SPS. The exact result returned depends on how many copies of each action are used to form the archetype, and this in turn depends on the distance from the new situation to each of the previous situations.

3.2.2. Rehearsal

If a word, w, is written more than once at an address, a, the lists at each of the responding addresses contain more and more copies of w. Therefore, the chances of recalling w increase, both when reading at a and at addresses near a. Human performance improves with practise, and the same is true of SPS. When a word is written many times, it is more likely to be recalled correctly, and more easily recalled at addresses near the write address. A word written many times will have more influence in the formation of the archetype, and thus we can move farther away from the original write address before we are unable to read the original data.

3.2.3. Recall Certainty

Assume that we read from a completely randomised memory: the archetype will be formed from a random collection of words. We can therefore expect the bit sums to be near the norm ($l/2$, where l is the number of words in the pooled data). This is a simple consequence of the words forming a binomial distribution. Furthermore, the probability that the bit sums are near the norm increases as l grows. Similarly, the chances of a bit sum being 0 or l rapidly diminish as l grows. For example, the chances of a bit sum being l (i.e., all ones) when $l = 50$ is approximately 1 in 10^{15}. Therefore, if a bit sum is equal to l, this strongly indicates that a 1 was actually written there. Thus, bit sums can be used not only to form the archetype, but also to give an indication of the certainty of the archetype. This property was mentioned briefly in Kanerva (1984), but not pursued any further. One of the main problems in

developing ESPS was discovering how to calculate and manipulate these certainties (this is covered in chapter 4).

3.2.4. The Best Match Problem

We've seen that if a word is written many times, recall of the word is easier, and in fact the word can be recalled even when reading at an address other than the original write address. This is the basis of solving the best match problem. SPS has found the data stored with the address that best matches the read address. Or, looking at it in terms of <*pattern*,*class*> pairs, SPS has found the class paired with the most similar stored pattern.

SPS can only solve a limited version of the best match problem. The form of the best match problem which SPS can solve was given in chapter 1 and will be reiterated here.

Assume that the set of pairs $\{<a_0, d_0>, \ldots, <a_k, d_k>\}$ have been stored in memory, and that we want to find the best match of an address a. Then, if a_0 through a_k are sufficiently different from one another, and if a is sufficiently similar to some a_l in the set of pairs, then reading at a will return d_l. The definition of "sufficiently similar" depends on the configuration of the memory, the composition of the data set, and the value of a.

We are now in a position to understand why these restrictions are made on the best match problem. First, the condition that a_0 through a_k be sufficiently different is necessary to ensure that interference doesn't destroy the data stored. If any two of the addresses are sufficiently similar, then the data written at one will be overwritten by the data written at the other.

Second, the condition that a be sufficiently similar to some a_l is necessary to correctly retrieve d_l. For example, if a is different enough from a_l that none of the actual locations at which d_l was stored respond to a, SPS will not retrieve d_l, except by blind chance.

There are other reasons why SPS solves only a limited form of the best match problem. When many words are written into memory, old words are lost, overwritten by newer words. Therefore, if the data set is larger than the capacity of SPS, it will not produce correct answers in some cases. Similarly, older words that have not been completely lost will be more difficult to recall than newer words, since they will be more likely to have had some copies of themselves overwritten by younger words. This means that there is an implicit bias or weight given to younger words. If the true best match is an old word, a younger word may be recalled instead because of this bias.

3.3. Realisation of SPS with Neuron-like Components

The architecture for SPS, as realised with neuron-like components, is shown in figure 3.5. The meaning and function of each of the components will be explained shortly.

SPS has memory locations (these are the actual locations), which respond selectively to read and write addresses (three of these are shown in figure 3.5). At each location, data is stored, and SPS must be able to add to the data stored at a location, and read the data stored at a location. To read, data from all responding locations must be pooled together and the archetype formed. These requirements are easily met with neuron-like components.

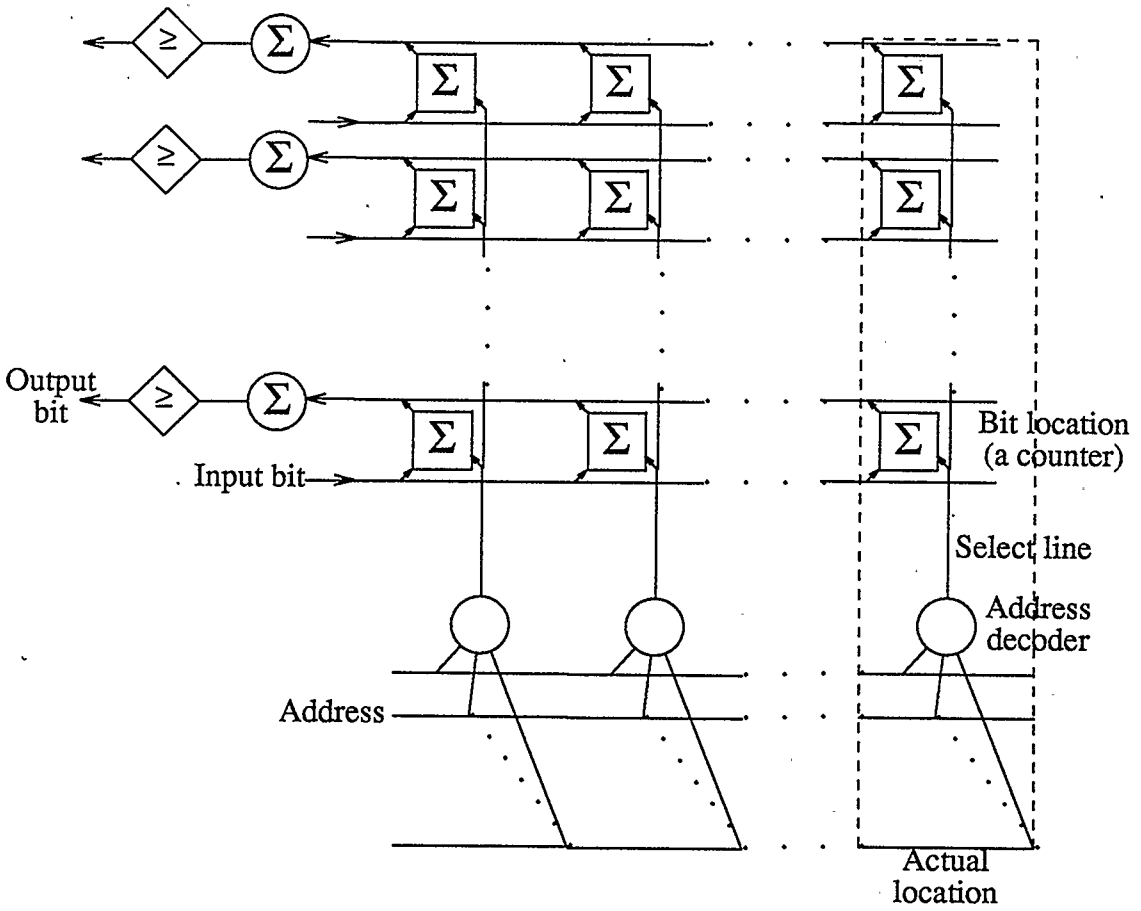


Figure 3.5. SPS Architecture. Adapted from Kanerva (1984).

3.3.1. Address Decoder Neurons

Each actual location has an address decoder neuron (three are shown in figure 3.5). This neuron responds to an address when the address is within *r* bits of the actual location's address. The address decoder neuron has *n* inputs, one for each bit of the address. These inputs can take on the values 0 or 1. On each input line there is an unchanging weight which has a value of either −1 or 1. The input on each line is multiplied by the weight on that line, and the *n* products formed are added together.

This sum is compared to a fixed threshold, and the result of this comparison (0 or 1), determines whether the neuron responds to the address or not.

An example address decoder is shown in figure 3.6. For this example, $n = 3$. The weights, w_0, w_1, and w_2, on the input lines are +1, −1, and −1, respectively.

Figure 3.7 shows the output of the neuron for all possible inputs, and thresholds (T) varying from $T = 2$ down to $T = -2$. We see that when $T = 2$, the neuron reponds to no address. When T is lowered to 1, the neuron responds to one address, 100. This is the address of the actual location. When T is lowered again, to 0, we see that the neuron responds to four addresses: 100, 000, 101, and 110. These addresses are all within 1 bit of 100 (its address). Thus, when $T = 0$, the value of r is effectively 1, since this neuron responds to all addresses within 1 bit of 100. Lowering the thres-

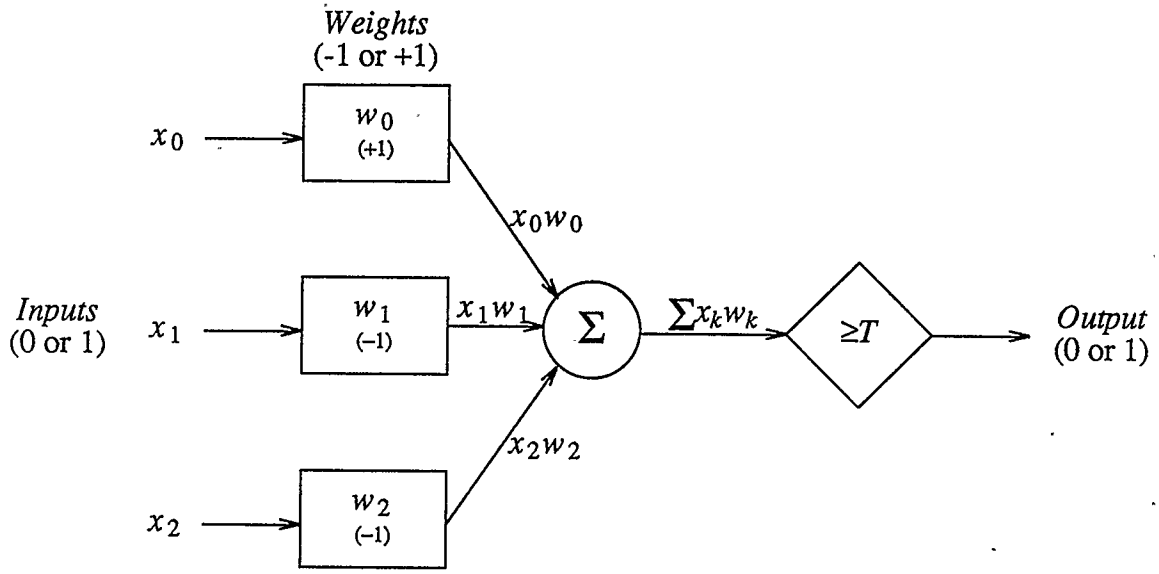


Figure 3.6. Address decoder neuron.

Input			*Output*					$d(100,Input)$
x_0	x_1	x_2	$T=2$	$T=1$	$T=0$	$T=-1$	$T=-2$	
0	0	0	0	0	1	1	1	1
0	0	1	0	0	0	1	1	2
0	1	0	0	0	0	1	1	2
0	1	1	0	0	0	0	1	3
1	0	0	0	1	1	1	1	0
1	0	1	0	0	1	1	1	1
1	1	0	0	0	1	1	1	1
1	1	1	0	0	0	1	1	2

Figure 3.7. Behaviour of address decoder neuron.

hold again to $T=-1$ effectively increases the value of r to 2, so that this neuron responds to seven addresses - all those within 2 bits of 100. These results are detailed in figure 3.7.

3.3.2. Storage Locations

Previously, each actual location was described as having a list of words. Managing a queue is a difficult task for a neuron, or group of neurons. Therefore, a slight amendment is made to the model. Instead of storing words in a list, where writing a word inserts the new word at the head of the list and removes the oldest from the end of the list, bit sums are stored. A bit sum will be stored at a modifiable weight between the output of the address decoder neuron and the input of an output neuron (see figure 3.5). At each location there are n counters, one for each bit. When a word is stored, counter k is incremented when bit b_k is 1, decremented otherwise. Since the storage capacity of a counter cannot be infinite, upper and lower limits exist on the counter. Incrementing a counter at its upper limits has no effect, as does decrementing a counter at its lower limit.

3.3.3. Output

When reading, the bit sums at each responding location are added up and thresholded. The only difficulty is in deciding the value of the threshold, since for a given read address, the number of responding addresses vary. Kanerva (1984) suggests using the mean bit sum over all the data stored for the threshold. He doesn't elaborate further, and it is difficult to imagine how this could be accomplished with neuron-like components.

An alternative scheme is used by Keeler (1987) and Chou (1987), where the threshold is fixed at 0, and counters are assumed to be capable of assuming positive and negative values (Kanerva assumed a lower limit of 0 for the counters). If the total sum is less than 0, the output will be 0, otherwise it will be 1. If this scheme is to be implemented with real neurons, however, it runs into trouble. Counters will be modifiable weights between neurons, and according to this scheme must be able to assume positive and negative values. However, Kanerva (1984) states that, as far as is known, weights do not change from excitatory to inhibitory (i.e., from positive to negative values).

A possible scheme would be to fix the threshold at some value, k, where k is calculated from the average number of responding addresses (when n is large, it will not vary by a large amount), and the expected bit sum in each location. If we assume that uncorrelated data is stored (i.e., 1's and 0's will be equally likely), then the expected bit sum would be the average of the upper and lower limits of the counter. Testing would confirm or refute this scheme.

The architecture for three actual locations is shown in figure 3.5. It is important to note that when reading or writing, the only locations that take part in the operation are those whose address decoders respond. Note also that address decoders share ad-

dress lines, and storage locations share input and output lines.

3.4. SPS as a Cerebellar Model

SPS is presented in Kanerva (1984) as a model of the cerebellar cortex, the elaborately folded outer layer of the cerebellum (Llinás, 1975). The cerebellum is intimately involved in motor control. Evidence of this is outlined in Arshavsky et al. (1986), where five observations are listed on the role of the cerebellum in motor control. They are:

(1) Removal of the cerebellum or its partial destruction results in motor disturbances. Movements are clumsy and slow.

(2) Cerebellar output signals reach all motor centres of the nervous system.

(3) The cerebellum receives signals from all motor centres as well as from the proprioceptors (a proprioceptor is a sensory receptor that responds to an internal stimulus, such as a muscle's position or tension).

(4) Stimulation of the cerebellum evokes various motor responses.

(5) The activity of cerebellar neurons is correlated with movements.

Kanerva's theory is similar to theories of the cerebellum put forward by Marr (1969) and Albus (1971). In particular, Albus developed a model of the cerebellar cortex called CMAC (Albus, 1979). CMAC was discussed in chapter 2. It has been used successfully to learn inverse dynamics equations for a robot arm, given a simple feedback system that generates approximations to desired trajectories, and a planning system that determines trajectories (Miller, Glanz, and Kraft, 1987). The addition of CMAC allows the arm to more precisely track trajectories. This is especially useful when arm speeds are high, since in such situations the feedback controller has difficulty following the trajectory. Given that the cerebellum is involved in coordina-

tion, this adds credence to CMAC (and therefore to SPS as well) as a model of the cerebellum.

However, the cerebellar theory proposed by Marr has of late fallen out of favour (Pellionisz, 1984), mostly due to lack of experimental results showing that synaptic change takes place at the connection between axons of neurons called *parallel fibres* and dendrites of neurons called *Purkinje cells*. Critical to the theories of Marr, Albus, and Kanerva is the modification of the strength of this connection in response to errors in its output compared to desired output. The state of Marr's theory is discussed by Pellionisz (1984). In spite of this evidence against these theories, however, the results are not conclusive, with one major school of experimentation still using Marr's theory as its foundation. For this reason I have continued to consider SPS a cerebellar model, and that, by extension, ESPS is as well.

3.5. Conclusions

In this chapter, we've seen how SPS can memorise <*address*,*data*> pairs. The manner in which it does so gives it several useful properties, including the ability to solve a restricted form of the best match problem. The specific scheme used by SPS can be implemented with neuron-like units, where these units and the way they are connected correspond to structures found in the cortex of the cerebellum.

However, as pointed out in chapter 1, the ability to memorise <*address*,*data*> pairs, even coupled with the ability to solve the best match problem, is insufficient when the learning system must solve a task within an environment that offers low-quality feedback. To deal with such an environment, the learning system must be able to generate and test its own solutions, and assign credit to steps in the solution. The changes made to SPS to produce ESPS, a learning system which has these abilities, are described in the next chapter.

CHAPTER 4

Extended Self-Propagating Search

We saw in chapter 2 that SPS is a supervised system, and as such is incapable of solving reinforcement problems. This chapter describes ESPS, an extension of SPS, which can solve reinforcement problems. The changes made to ESPS enable it both to generate new solutions and assign credit to steps in the solution. As established in chapter 2, these are necessary qualities for reinforcement learning systems.

This chapter starts with an overview of the operating procedure of ESPS. This gives the reader a framework into which the details, explained in later sections, can be fitted. Next, the changes and additions made to SPS are presented, and it is argued that these changes make ESPS a reinforcement learning system. Finally, physiological justification is provided for the changes made to SPS.

In this and subsequent chapters, it is assumed that properties of SPS also apply to ESPS, unless otherwise noted. Therefore, when talking about some property of SPS and ESPS, reference will be made only to SPS, with the understanding that these properties apply to ESPS also. When a property is unique to ESPS, then only ESPS will be referred to.

4.1. Operating Procedure

This section describes the operating procedure of ESPS. The operating procedure is a short sequence of steps which are repeated until the goal is reached. During each iteration, ESPS decides on the action to be performed in the current situation, which is then executed. Reinforcement from the environment, if any, is re-

ceived, and memory is updated according to the reinforcement.

So, each iteration of the operating procedure consists of the following two steps:

(1) Action Generation and Execution

(2) Reinforcement and Memory Updating

These steps are discussed in detail later in the chapter. The rest of this section discusses the properties of SPS, and how they overcome the problems posed by the size of the situation space.

4.1.1. Coping With Large Situation Spaces

For a problem of reasonable complexity, the number of possible situations is very large. This has two implications. First, not all situations can have their own "slot" in memory - ESPS, or any system, is restricted to storing some subset of possible situations. Second, there must be some mechanism for generalisation. This is due both to the size of situation space, as it would take too long to derive the correct action for each situation, and the restriction to a subset of situations, since actions for situations which have no "slot" must be induced from other known <*situation*,*action*> pairs.

SPS can handle the subset restriction. We saw in chapter 3 that for SPS to store items in an n-dimensional address space, it only needed a randomly chosen subset of addresses, rather than all 2^n addresses.

As explained in chapter 3, generalisation is a natural emergent property of the storage scheme. If action a is written at situation s, then reading at a situation close to s (call it s') will return an action close to a (call it a'). The implication of this property is that ESPS does not have to store an action for each situation. Situations not directly written to will be automatically "filled in" with an action that is some aver-

age of nearby actions. This lessens the amount of work ESPS has to do, and means that it can produce good results in novel situations - assuming that it wants to do similar things in similar situations. This assumption is made, and is a critical determinant of the types of problems ESPS can solve. ESPS cannot solve a problem in which two situations *which have similar representations* have radically different actions associated with them. In order for two situations to have different actions associated with them, they must be given dissimilar representations.

4.2. Changes to SPS

Functionally, ESPS must be able to do two things that SPS cannot. First, it must be able to generate new solutions. This is accomplished by changing SPS's deterministic reading process to a non-deterministic one. Making reading non-deterministic means that ESPS is not restricted to only producing actions it has recorded, but is capable of generating new ones.

The second requirement is that ESPS properly assign credit to steps in the solution. This is done through *expectations* and an extended concept of reinforcement. In ESPS, each <*situation*,*action*> pair has associated with it a number which is its expectation of future reinforcement if *action* is executed in *situation*. Reinforcement, in turn, can come not only from the environment, but also from the expectations of future steps.

Expectations are updated according to the reinforcement received. When reinforcement is different than what is predicted by the expectation, then the expectation stored with the <*situation*,*action*> pair is updated to make it better fit reality. The goal of this process is for each <*situation*,*action*> pair to have associated with it an accurate prediction of future reinforcement. To implement such a scheme, ESPS must be able to associate an expectation with each pair, and be able to manipulate

those expectations. The changes made to SPS to produce ESPS are summarised in figure 4.1.

Extensions to SPS		
Requirement	Extension	Physiological Justification
Generating new solutions	Non-deterministic reading (f function). The f functions tested are: $$f_B(k,l) = \frac{\sum_{i=0}^{k} \binom{l}{i}}{2^l}$$ and $$f_L(k,l) = k/l$$ Instead of maintaining an explicit action, probabilities of actions are stored. The probabilities are manipulated by changing the bit sums (k).	Imperfect neurons.
Credit Assignment	Storage and manipulation of expectation, and an extended notion of feedback. Expectations are calculated according to $$E = \frac{\sum_{k=1}^{n} b_k p_k + (1-b_k)(1-p_k)}{n}$$ When feedback for an action exceeds its expectation, that action is made more probable by manipulating the bit sums (k). The notion of feedback is extended to include not only environmental feedback, but also the expectations of subsequent steps.	Global access to expectations and feedback through climbing fibres.

Figure 4.1. Extensions to SPS.

4.2.1. Non-Deterministic Reading

Non-deterministic reading is a bit-wise process - calculation of each bit of the archetype is done independently of the others, as was the case with SPS. Therefore, the following discussion refers only to the calculation of a single bit of the archetype, with the assumption that this process is applied to all bits of the archetype. Further, it is assumed that the pooled data contains l words, and so l bits are being used to form each bit of the archetype, and that k of them are 1's (and therefore $l-k$ are 0's). The value k will be referred to as the *bit sum* since it is equivalent to the sum of the l bits.

The value of a given bit of the archetype is determined by applying a function, f, to the bit sum, k. The value of the function is the *probability* that the output bit will be set to 1 for a given k. A probability of 0 means that the archetype will be 0 at that bit position, while a probability of 1 means that the archetype will be 1. Values near 0 or 1 mean that the archetype will be, with high probability, 0 or 1, respectively. A value of 0.5 means that the archetype will be 0 or 1 with equal probability.

The probability returned by the f function indicates ESPS's certainty of its answer. When the probability is at or near 0 or 1, certainty is high. When the probability is at 0.5, uncertainty is at its maximum.

The f function is defined to be monotonically increasing with k:

$$f(k_1, l) \leq f(k_2, l) \text{ iff } k_1 \leq k_2$$

This means that to increase the probability that the given bit is 1, the bit sum (k) should be increased. This is done by writing a 1. To increase the probability that the bit is 0, k should be decreased, which is done by writing a 0. This provides a simple mechanism for positively or negatively reinforcing a result - if reading results in a value v for a bit b, then to positively reinforce b (i.e. make it more likely to occur in the future), v should be written. Similarly, to negatively reinforce b (i.e. make it less

likely to occur in the future), the inverse of v (denoted $\tilde{v}$) should be written.

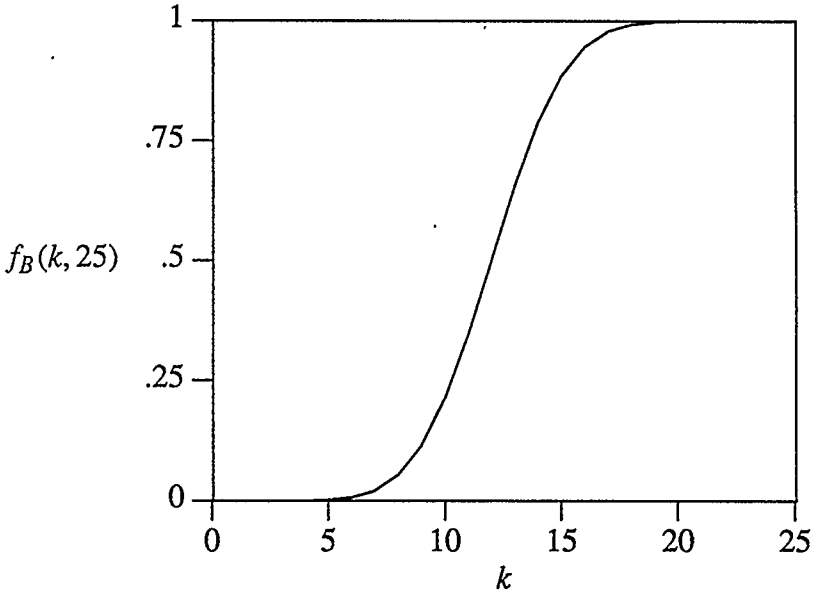
Non-deterministic reading provides a method of controlling the search done by ESPS for the correct solution. When search is to be limited, as is the case when a good action is found, 0's or 1's, as appropriate, should predominate in each bit of the pooled data. When search is to be unconstrained, 0's and 1's should occur so as to make f return 0.5.

The f function has two arguments, k (the bit sum), and l (the number of words in the pooled data). I have not been able to find a "best" f function, nor is it clear that such a function exists. Two certainty functions were tested, one based on the binomial distribution (f_B), the other a simple linear rule (f_L). The first, f_B, was chosen arbitrarily, on the basis of intuition. I have not been able to justify its use (other than that it works). The second, f_L, was chosen in response to my inability to justify f_B. That is, it was chosen to answer this question: if I cannot prove there is a "best" function, or justify the choice of f_B, does this mean that ESPS will work with any function which satisfies the condition of monotonicity?

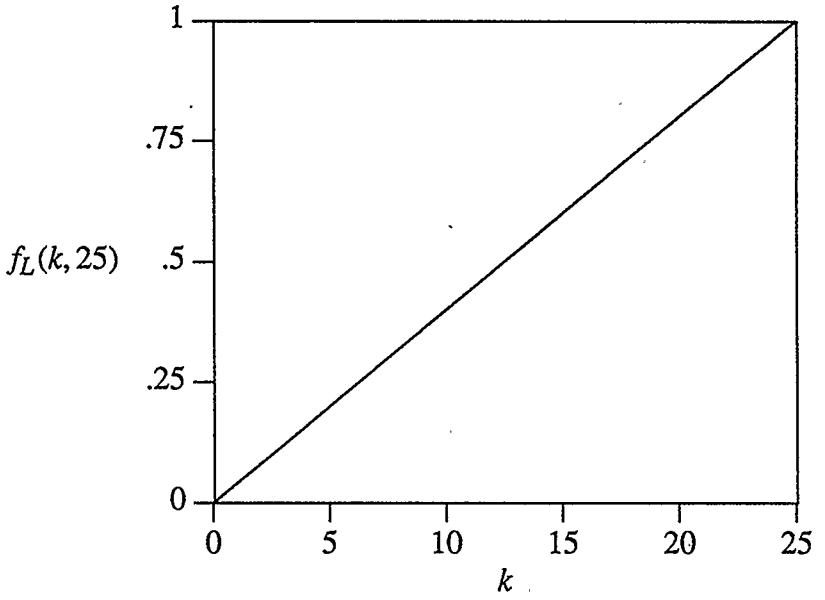
The first certainty function is:

$$f_B(k,l) = \frac{\sum_{i=0}^{k} \binom{l}{i}}{\sum_{i=0}^{l} \binom{l}{i}} = \frac{\sum_{i=0}^{k} \binom{l}{i}}{2^l}$$

This function expresses the probability of there being k or fewer 1's in a randomly chosen set of l 1's and 0's. When $k = l$, $f_B(k,l) = 1$, when $k = \lfloor l/2 \rfloor$, $f_B(k,l) = 0.5$, and when $k = 0$, $f_B(k,l) = \frac{1}{2^l}$. The last value, $f_B(0,l)$, is therefore not 0, as might be expected, but does tend towards 0 as the sample size, l, tends to infinity. A graph of f_B for $l = 25$ is shown in figure 4.2(a).



(a) Behaviour of $f_B(k,l)$ for $l = 25$.



(b) Behaviour of $f_L(k,l)$ for $l = 25$.

Figure 4.2. Behaviour of f_B and f_L.

The second certainty function is:

$$f_L(k,l) = k/l$$

Its behaviour for $l = 25$ is shown in figure 4.2(b). This function was chosen to test the sensitivity of ESPS to different f functions. The results of using each function are discussed in chapter 5.

The function used by SPS (the majority rule) can be recast as an f function. The function is:

$$f_M(k,l) = \begin{cases} 1 & \text{if } k \geq l/2 \\ 0 & \text{otherwise} \end{cases}$$

Since the probabilities returned by f_M are 0 and 1, f is a deterministic step function.

4.2.2. Expectations

As mentioned in section 4.2, credit assignment is done through the use of expectations and an extended definition of reinforcement. If the *<situation,action>* pair p has stored with it a number which accurately reflects future reinforcement, then pairs leading to p need not wait for environmental reinforcement. By examining the expectation associated with p, they can immediately determine the value of their action.

Expectations are learned, so they must be updated when expected reinforcement does not match actual reinforcement. The goal is for expectations to accurately predict actual reinforcement, thereby making rewards and punishment based on those expectations correct.

Expectations are stored via the bit sums. A *<situation,action>* pair whose expectations for future reinforcement are high will be stored such that the f function will return a value near 0 or 1. That is, it will be stored such that it will be recalled with high certainty. Aside from recording expectation, this has the effect that *action* will be recalled with little change when *situation* is encountered in the future. This is

a desirable effect. When expectations are high, the search for better solutions should be narrowed, as the current one already leads to high reinforcement. When expectations are low, the search for better solutions should be wide-ranging, since the ones already found did not lead to high reinforcement.

Because the functions of search and storage of expectation are combined into a single mechanism, the range of values that reinforcement can take are important. If reinforcement does not reach a high enough value, then actions will not be stored with high enough certainty. So, even if an action maximises available reinforcement, it will not be recalled with high certainty, and ESPS will continue to search for a better solution, one which does not exist.

This is in contrast to ASN and ASE/ACE, where certainties are stored via the weights in the output neurons, while expectations are stored in the predictor neuron. Under the ASN and ASE/ACE scheme, certainties and expectations can be manipulated independently, and thus there is no restriction on the values that reinforcement can take as there is in ESPS. This flexibility is bought at the cost of an extra component. Combining the two functions together as in ESPS creates a simpler solution, and one more easily justified on physiological terms.

As a final point, note that ESPS cannot represent an expectation of negative reinforcement for a given action. Actions which lead to negative reinforcement can be made less likely to occur only by making other actions more likely to occur.

4.2.2.1. Calculating Expectations

Expectation is a measure that applies to an entire word. However, the probabilities returned by the *f* function are defined only for individual bits. Therefore, it is necessary to be able to take individual bit probabilities and calculate the expectation for the entire word.

Expectations are calculated according to the following formula:

$$E = \frac{\sum_{j=1}^{n} b_j p_j + (1-b_j)(1-p_j)}{n}$$

where E is expectation, p_j is the probability that bit j will be 1, $(1-p_j)$ is the probability that bit j will be 0, and b_j is the actual value given to bit j. For each bit, the probability of the bit taking on its actual value is calculated (this is $b_j p_j + (1-b_j)(1-p_j)$). The average of these probabilities becomes the expectation.

Note that the calculation of E is unusual in that it is dependent on the actual values given individual bits. One might expect E to be just the average of the bit probabilities. However, it was felt that E should be a measure of the expectation of reinforcement for the *actual* word produced, rather than of the most probable. If all of the p_j were 0.9 (i.e., each bit is likely to be 1), yet the resulting word was all 0's, then E would be 0.1, because ESPS's expectation of reinforcement for that particular word, on the basis of the bit probabilities, should be low. This function has the property that the most probable word produces the highest expectation, while the least probable produces the lowest expectation.

As with the f functions, I have no proof that this method of calculating is optimal. It may be that other methods of calculating E are better. However, only this particular one has been tested.

4.2.2.2. Feedback Scheme

To cope with the infrequent feedback offered by the environment, ESPS extends the concept of feedback. Feedback, instead of just originating from the environment, can also originate internally, in the form of the expectations of other steps. If an action, a, has a high expectation for future reinforcement, other actions which lead to a

should also be given high expectations. The ultimate source of these high expectations is the environment. Steps which lead directly to environmental feedback have their expectations set according to that feedback. All other steps receive this feedback indirectly through other step's expectations.

The feedback scheme will first be illustrated from the point of view of a bit location (recall figure 3.5). This is done to make clearer the relationship between ESPS and the systems in chapter 3. Thus, for the purposes of this explanation, we assume that bit sums are stored, rather than a list of words.

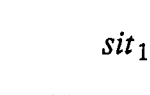
Recall that when the address line for a bit location is active, the value contained in the bit location, w, is placed on the output line, as are the values contained in all other activated locations on that line. The f function is applied, producing an output, y. If feedback exceeds expected feedback, then w should be updated so as to increase the probability of generating y in the future. Thus, if $y = 1$, then w should be incremented, and if $y = 0$, w should be decremented.

The feedback rule thus is:

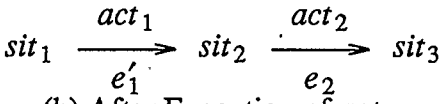
$$w(t+1) = w(t) + greater(F(t+1), E(t))(2y(t)-1)$$

where $w(t)$ is the value in the bit location at time t, $F(t+1)$ is the feedback at time $t+1$, $E(t)$ is expectation at time t, $greater(x,y)$ is 1 when $x \geq y$, -1 otherwise, and $y(t)$ is the output at time t. Note that the feedback for an action at time t is received at time $t+1$. Also, the value of $F(t+1)$ is the value of environmental feedback when environmental feedback is present, otherwise it is the expectation of the next step.

Figure 4.3 shows how the expectation of a single *<situation,action>* pair ($<sit_1, act_1>$) is updated according to the expectation of the subsequent pair ($<sit_2, act_2>$). Figure 4.3(a) shows act_1 being performed in response to encountering sit_1. The process of reading act_1 generates an expectation of reinforcement, e_1. Ex-



(a) After Execution of act_1.



(b) After Execution of act_2.

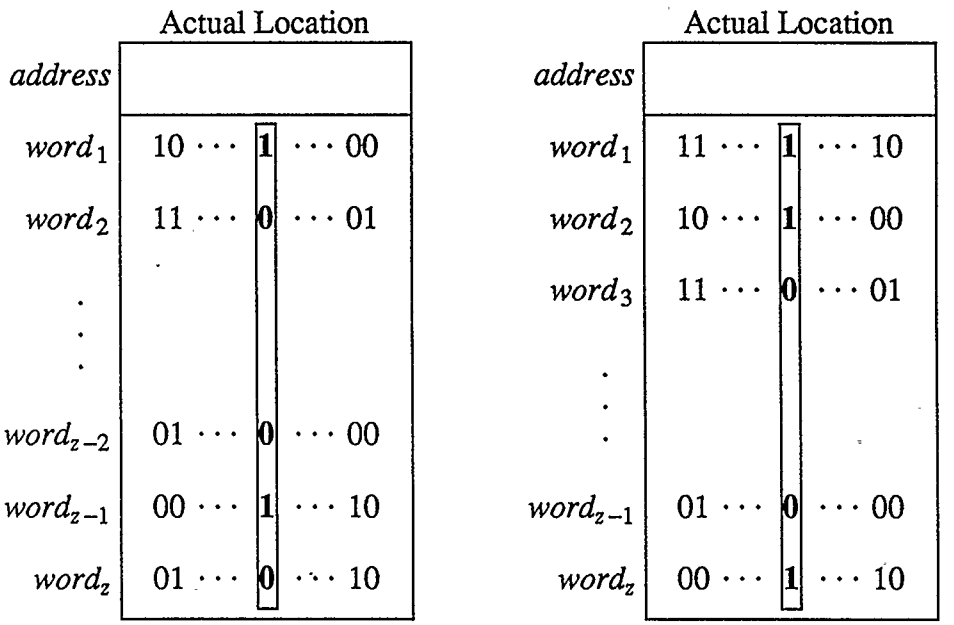
Figure 4.3. Updating Expectations.

ecuting act_1 in situation sit_1 leads to sit_2. The value e_1 is not updated (becoming e_1') until the feedback of the next step is received. Since this is a reinforcement problem, feedback is not received on every step, which is the case here. However, $<sit_2, act_2>$ has associated with it an expectation of future reinforcement, e_2. (figure 4.3(b)). Once e_2 is available, e_1 can be updated. If $e_1 > e_2$, this means that the expectation of the first pair is too high - act_1 led to a situation which was worse than expected. The value e_1 should be lowered. If $e_1 < e_2$, this means that the expectation of the first pair is too low - act_1 led to a situation which was better than expected. The value e_1 should be lowered.

Note that the process of updating e_1 changes the probability that act_1 will be produced in the future. If e_1 is increased, reading will become more deterministic, increasing the chances of producing act_1 in the future. This is desirable, since e_1 is increased only if act_1 leads to a situation which is better than expected. The opposite is true when e_1 is decreased. The probability of producing act_1 is decreased, which is desirable, since act_1 led to less reinforcement than expected.

From the point of view of the storage scheme actually used by ESPS, the feedback rule slightly more complicated. The difference, as explained in chapter 3, is that SPS (and therefore ESPS), stores explicit words, instead of n bit sums. Thus, instead of having a weight w for each bit, there is a list of bits, where the length of the list is the number of words stored per address. Figure 4.4(a) shows an actual location at which z words are stored. Bit b of each of these words is highlighted.

Bit b of the output ($y_b(t)$) is determined by adding together bit b from each word at each responding address, then applying the f function. If feedback exceeds expected feedback, then this bit sum should be updated so as to increase the probability of generating y_b in the future. Thus, if $y_b(t) = 1$, then the bit sum should be increased,



(a) Before Reinforcement (b) After Reinforcement

Figure 4.4. Feedback in ESPS.

and if $y_b(t) = 0$, then the bit sum should be decreased.

To increase the bit sum for bit b, a 1 is written at bit b in each responding actual location. To decrease the bit sum, a 0 is written. This is shown in figure 4.4(b). Note that writing a 1 does not guarantee that the bit sum will increase, nor does writing a 0 guarantee that the bit sum will decrease. Writing adds a new bit to the list, displacing the oldest. If the new bit and the oldest bit have the same value, the bit sum will be the same. On the other hand, if the new bit and oldest bit have different values, writing will change the bit sum by two (this is the case in figure 4.4(b)). Overall, assuming random data, the average change will be 1.

If we now examine the process from the standpoint of an entire word, the process of reinforcement becomes quite simple. If reading produced **y** and positive reinforcement is called for, then **y** is written. If negative reinforcement is called for, then ~**y** is written, where ~**y** is **y** with every bit flipped.

ESPS's feedback scheme is analogous to Holland's approach to credit assignment, where classifiers which lead to other useful classifiers will have their strengths increased. It is also closely related to the scheme used by ASE/ACE, where the element responsible for predicting feedback (ACE) updates its predictions on the basis of environmental feedback and on the predictions of subsequent steps.

The concept of expectation also appears in Andreae's work on Multiple Context Learning Systems (MCLS's) (Andreae, 1972, Andreae, 1977, Andreae and MacDonald, 1987). Expectations have a slightly different meaning in MCLS's, reflecting the probability of an action leading to a goal state (goal states receive reinforcement), rather than the expected reinforcement received if that action is executed. Expectations are highest for those actions which are part of the shortest path to the goal. That is, the "best" solution in an MCLS is the shortest one which leads to reinforcement

(called "reward" by Andreae).

Expectations are updated via a process termed "leakback", in which the expectations of future steps are propagated backwards through previous steps and recalculated with Howard's policy iteration method (Howard, 1966). Because of the structure of the memory of MCLS, it is possible to perform this process without requiring the system to execute in the environment. That is, it can update expectations internally. Also, because expectations are stored separately, they can be updated without affecting the stored solutions with which they are associated. The accuracy of the leakback process depends on the accuracy of its internal model of the environment - the generated expectations will be meaningful if the internal model is accurate.

ESPS is meant to retain the physiological plausibility of SPS. Therefore, the changes made to ESPS must be justified not only on a functional basis, but also on a physiological basis. The next section provides that justification.

4.3. Physiological Justification

Three things must be justified:

(1) A mechanism for non-deterministic reading.

(2) Having global access to environmental feedback.

(3) Having global access to expectations.

4.3.1. Non-Deterministic Reading

The premise of non-deterministic reading is that successive reads at a given address will return different results. The justification for such a process in the cerebellum is the assumption that neurons are imperfect. In Kanerva (1984), the comparison between a bit sum and its threshold is postulated to be performed through the combined actions of three neurons - Purkinje cells, basket cells, and stellate cells. The

Purkinje cell is responsible for forming the bit sum and performing the comparisons, while the basket and stellate cells adjust the threshold of the Purkinje cell. If these neurons are perfect, then the comparison between the sum and the threshold will be performed correctly each time, and so reading will be deterministic.

However, it seems more reasonable to assume that neurons are imperfect, so that the comparison is not always correct. That neurons are not perfect (i.e. that they are not deterministic) is a well-established fact. In Sejnowski (1986, pg. 378), he states:

> the responses of single neurons in cortex often vary from trial to trial ... Therefore, in many experiments the spike train is averaged over a number of responses (typically 10) to obtain a post-stimulus time histogram. The histogram represents the *probability* of a spike occurring during a brief interval as a function of time following the stimulus.

(emphasis mine).

Given that neurons are imperfect, and that the neurons in question (Purkinje cells, basket cells, and stellate cells) perform a thresholded weighted sum, it seems reasonable to assume that the probability of error increases as the value of the sum approaches the threshold. That is, when the bit sum is near the threshold, the presence of noise could easily shift the result from one value to another. When the bit sum is far from the threshold, the presence of noise will have less of an effect.

This is the behaviour produced by the f functions. When the bit sum equals the threshold, 1's and 0's are equally possible. As the bit sum moves away from the threshold, one value is produced with increasing probability.

Thus, by assuming imperfect neurons, we can see how the cerebellum can produce different outputs in identical situations. Moreover, the variation in outputs depends on the bit sums - the more extreme the bit sums, the less the variation in output.

4.3.2. Global Access to Environmental Feedback

There are two possible explanations here: one is a chemical mechanism, whereby some chemical change prompted by environmental feedback changes the characteristics of storage. While possible, it would be slow, especially considering the relatively high speed at which the cerebellum operates (Blomfield and Marr, 1970).

The second possibility is that interneurons of some sort signal the presence and strength of feedback. There are neurons, called climbing fibres, that originate outside of the cerebellum which synapse with all Purkinje cells (Purkinje cells are the cells which store the bit sums). These climbing fibres have a powerful excitatory effect on the Purkinje cells they synapse with. The theory postulated here is that when a Purkinje cell takes part in the producing an output, it may have a period of eligibility in which its weights can be updated. The update is performed on the basis of the feedback received through the climbing fibres during that time. This explanation is particularly appealing because it neatly accounts for the function of climbing fibres, which were previously postulated to supply the correct action, when the cerebellum was considered a supervised system.

4.3.3. Global Access to Expectations

Once again, the same two possibilities apply. The chemical mechanism would work the same here as for environmental feedback. The interneuron mechanism would require an interneuron which distributes the signal from a Purkinje cell (where the expectation would be generated) to all other Purkinje cells. There is no neuron of this type in the cerebellum. There are interneurons that synapse with Purkinje cells which could distribute its signal locally, but not globally. So one is forced to conclude that either only local expectations are available, or that the expectations generated by Purkinje cells are propagated by a more circuitous route, out of the cortex

and then back in again, perhaps by climbing fibres.

CHAPTER 5

Experimental Results

This chapter describes the implementation of ESPS, both as it is implemented on a single processor, and as it is implemented as a distributed system over many processors. Also described in this chapter are the experiments that were run on ESPS, and the results of those experiments. The chapter concludes with a discussion of the experimental results.

5.1. Structure of ESPS Implementation

ESPS consists of a set of c actual locations, each containing a list of p m-bit words. Addresses are n bits long. ESPS is implemented by a c element array, where each array element contains 1 n-bit word (the address) and p m-bit words (the words stored at that location). This is shown in figure 5.1.

Writing the word w at address a consists of adding w to the lists at all responding actual locations. Adding w to the list at a responding address means shifting all words to the right (thus losing the rightmost, and oldest, word), and placing w in the leftmost slot.

Responding locations are all those actual locations whose addresses are within r bits of a (r is the read/write circle size). Finding the responding addresses means comparing each actual address with a. Thus every element of the c element array must be checked. This is true for writing and for reading.

Reading at a requires maintaining a count, l, of the number of words used to form the archetype. This will be a product of the number of responding addresses

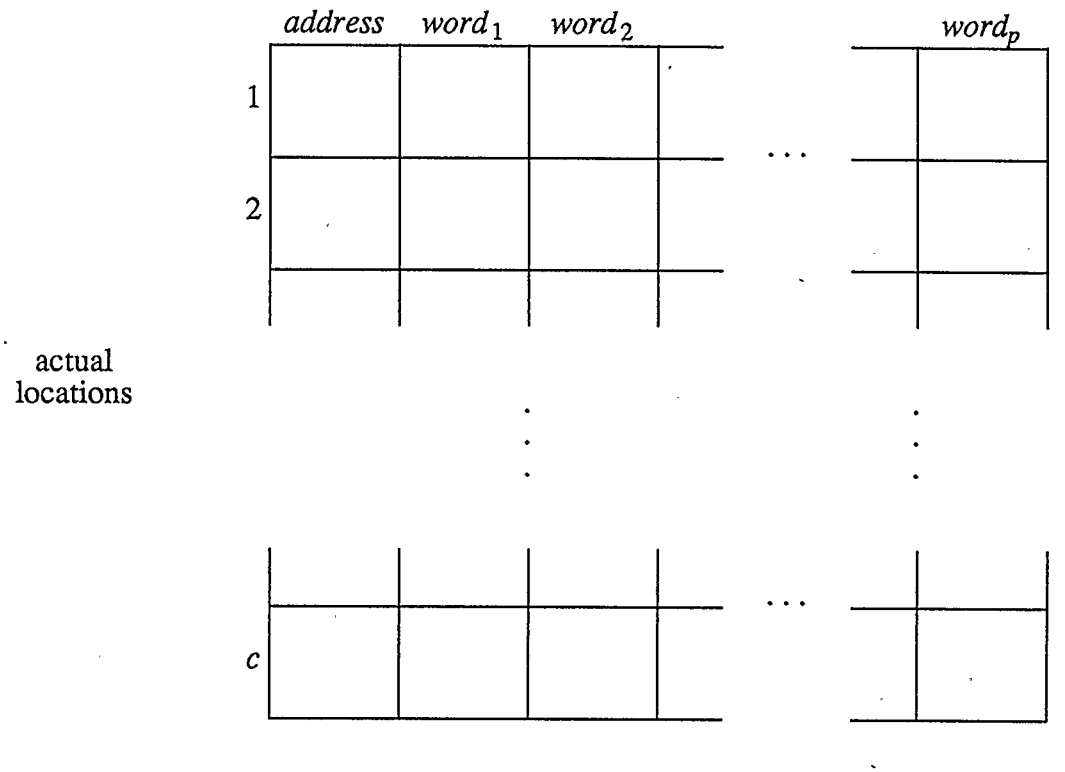


Figure 5.1. Structure of ESPS.

and the number of words per address (p). Also needed is an array of m counters, one per bit of the archetype. This array is initially set to 0's. For each word in the list of each responding address, counter k is incremented if bit k of that word is 1.

After all responding addresses have been checked, the archetype is formed. Formation of the archetype depends on the f function used. The deterministic step function used by SPS (f_M) compares each counter to the value $l/2$. If counter b is less than $l/2$, then bit b of the archetype is 0, otherwise it is 1.

The binomial f function (f_B) sets bit b of the archetype to 1 with the following probability

$$f_B(k,l) = \frac{\sum_{i=0}^{k/d} \binom{l/d}{i}}{\sum_{i=0}^{l/d} \binom{l/d}{i}} = \frac{\sum_{i=0}^{k} \binom{l/d}{i}}{2^{l/d}}$$

where k is the value of counter b. Due to the limitations of floating point numbers on the computers, the values k and l were divided by a constant, d, in the calculation of f_B. This prevents floating point overflows from occurring when calculating 2^l. It is possible that the division by d qualitatively changes the performance of ESPS - the properties of f_B will change, having a steeper slope in the central section. Nevertheless, the basic properties of f_B remain - it remains non-linear, with a rapid change in value as k passes through the value $l/2$ (see figure 4.2).

The linear f function (f_L) sets bit b of the archetype to 1 with probability k/l.

5.1.1. Structure of Distributed Implementation

A typical memory configuration has 10,000 actual locations, with anywhere from 10 to 90 words per address. A write involves looking at each actual address to see if responds to the write address. If it does, all the words are shifted one place, and the new word added. A read also involves looking at each actual address. Those that respond have their data pooled together. With a large memory, these are time-consuming operations. However, ESPS is easily distributed over several processors, with a nearly m times speedup, where m is the number of processors.

Consider a memory with one processor per actual location. To write word w at address a, one need only send a copy of w and a to each processor. Each processor compares a with the address of its actual location, and if the difference is less than r bits, writes w. This distributed approach is feasible because each processor can act independently of the others.

Reading is slightly more complicated than writing, as the data from all responding addresses must be gathered together. Each processor generates a vector of m counters, as well as recording how many words were gathered to form the vector. The vectors from all processors are added up, as are the word counts. These totals are then used to form the archetype.

In general, where P processors are available, each processor can be assigned c/P addresses. The description in the previous paragraphs assumed $P = c$, and the previous section assumed $P = 1$.

5.1.2. Jipc Implementation

Memory was distributed via Jipc, the Jade interprocess communications facility (Unger, Dewar, Cleary, and Birtwistle, 1986). Jipc is a synchronous message-passing system. Two processes communicate using a send/receive/reply sequence.

For example, assume that process p_1 wants to send information to process p_2. Process p_1 would create its message, placing it in its *message buffer*. It then sends the message buffer to process p_2. Until p_2 receives the message and replies to it, p_1 is blocked; it cannot do anything.

Process p_2 cannot access the message until it does a receive. If p_1 has not already sent the message and p_2 does a receive, then p_2 will be blocked until the message is sent.

Once p_2 has received the message, it is free to do whatever processing it likes, with p_1 blocked the whole time. The data sent by p_1 is accessible in p_2's message buffer. Once p_2 replies to p_1, p_1 will become unblocked and both will continue on.

ESPS is implemented as a set of P+1 Jipc processes on P processors. There are P server processes, one per machine, each containing c/P addresses. In addition,

there is one controller process which is responsible for organising the activity of the server processes. This is shown in figure 5.2.

When ESPS is started up, it is the controller process which is created. The controller process reads a file which determines the memory configuration. This file gives the size of addresses (n), the size of words (m), the number of words per actual location, the number of actual locations (c), and the read/write circle size (r). Having read this file, the controller then creates the server processes, using another

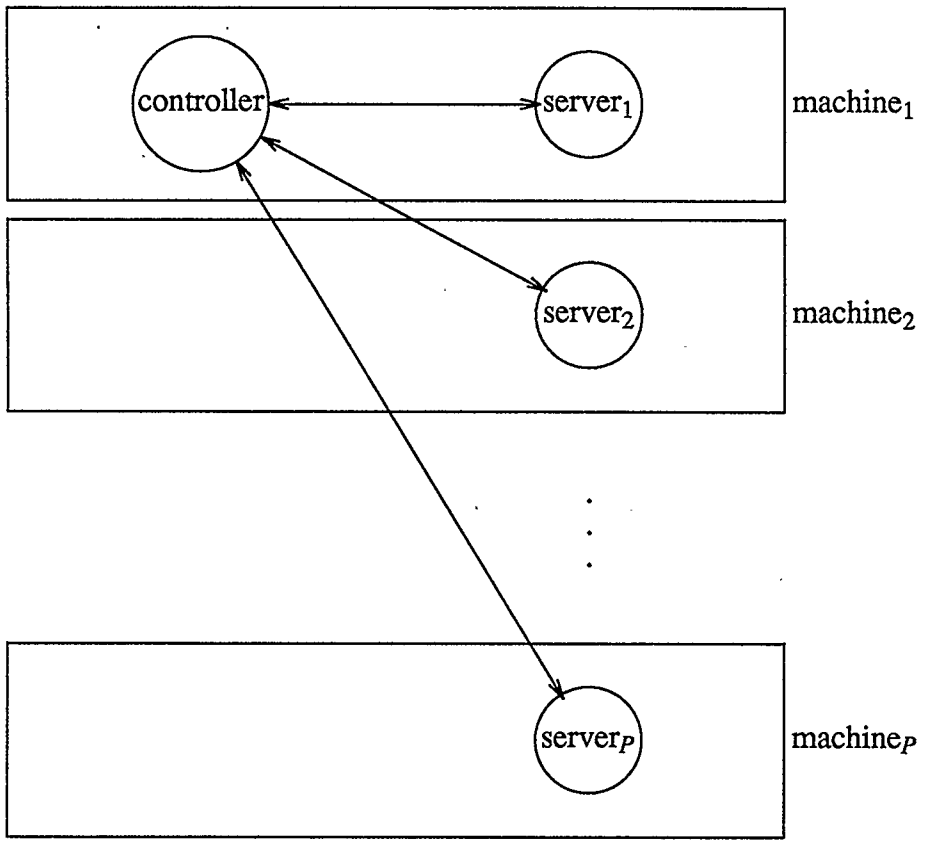


Figure 5.2. Distributed Implementation of ESPS.

configuration file to determine the machines on which these processes can be created. The memory is divided equally among all servers.

The servers initialise their own memories to random values, then do a receive from the controller, awaiting instructions. The instructions of interest here are writing data at an address and reading the data at an address.

Jipc buffers can contain typed data items, such as integers, floating point numbers, strings, and byte blocks (a byte block is an array of bytes). Furthermore, a buffer can contain several pieces of data at once, so it could contain an integer followed by two floating point numbers, for example. In the case of ESPS, messages from the controller start with an integer giving the instruction (**write** or **read**). Following this is the information necessary for the instruction.

When the instruction is **write**, the buffer will have two byte blocks, one containing the address, and one containing the data. Upon receipt of the message, the server process copies the data from the buffer and immediately replies. This frees the controller to send the **write** instruction to the other servers. After replying, the server then performs the write.

When the instruction is **read**, the buffer will have one byte block, namely the address. Like the write, the address is copied from the buffer and the server immediately replies. It then performs the read, creating two pieces of data. The first is an integer indicating the number of words which were read (l). The second is an array of m integers containing the bit sums. Once the server has finished its read, it places l and the array into the buffer, and sends it to the controller.

From the controller's point of view a read operation consists of sending all servers the read instruction along with the address, then performing a receive from each server in turn. The controller has its own l value and bit sum array, both initial-

ised to 0. As each server replies, the controller updates its l value and bit sum array by adding to them the information the server sends. After all servers have sent their information, then the controller forms the archetype.

5.1.3. Implementation Details

The code for ESPS and the test routines comprises about 2000 lines of C code, and is broken down as follows.

The code for distributed reading and writing is contained in three files, **sps.c**, **jipc_sps.c**, and **jipc_sps_server.c**. The file **sps.c** contains code for allocating and initialising memory, reading from and writing to that memory, and code for the various f functions. A system using ESPS, but only needing a single processor implementation, would directly use the routines in this file.

A system needing a multi-processor implementation would call the routines in **jipc_sps.c**. To a calling program, the routines in this file are identical to those in **sps.c**, except that the routine names have a **jipc_** prepended to them. Therefore, it is a simple matter to change a system from a single- to a multi-processor implementation. One merely needs to change the names of the routines, and recompile with the routines in **jipc_sps.c**.

The routines in this file take care of creating and destroying jipc server processes on other machines, and creating and gathering messages. The actual work of reading and writing is done in the jipc server processes. The code for the server process is contained in **jipc_sps_server.c**. This file contains the code for a stand-alone process. The server continually performs a loop in which it waits for a command, performs the command, then waits for the next command. The server uses the routines in **sps.c** to perform the actual acts of reading and writing.

The files, their contents, and their sizes are shown in figure 5.3. In addition, two files containing useful routines are used in ESPS. They are **bit_stuff.c** and **probability.c**. The first file contains routines for setting and reading individual bits in bit strings (bit strings are represented as character arrays). Also included in the file are routines for printing out and reading in bit strings, useful for debugging. The second file contains routines for generating random numbers.

The file **testk.c** contains code to perform the single- and multi-step tests. The parameters given to **testk** are contained in a file specifying word size, address size, number of actual locations, words per address, read/write circle size, number of trials, and the number of steps per trial.

ESPS Code Details

File	Contents	Length
sps.c	Code for allocating and initialising memory, reading from and writing to memory, on a single processor.	622 lines
jipc_sps.c	Code for allocating and initialising memory, reading from and writing to memory, on many processors.	344 lines
jipc_sps_server.c	Code for server process.	180 lines
bit_stuff.c	Bit string manipulation routines.	258 lines
probability.c	Random number generation.	113 lines
testk.c	Code for tests.	290 lines

Figure 5.3. ESPS Code Details.

When run, it creates the necessary processes and perform the test, dumping the results to a file with suffix **.trace**. In this file are the start and end time, and the results of each step in each trial, giving the expectation, the reinforcement, and whether the action or its inverse was written.

Typical runs involved up to four Sun Microsystem workstations, depending on availability. Run times varied, depending on the test length, the number of steps per trial, the size of the addresses, the size of the words, the number of actual locations, the number of words per location, the number of machines used, and the load on each machine. For example, a run involving 2500 trials, two steps per trial, 128-bit addresses and words, 10,000 actual locations, 50 words per location, and four Suns takes 7.5 hours. A run involving 100 trials, one step per trial, 128-bit addresses, 8-bit words, 10,000 actual locations, 10 words per location, and four Suns takes 5 minutes.

5.2. Experiments

Two experiments were run, the first being a single-step learning problem, the second a multi-step learning problem. The purpose of the first experiment is to establish that ESPS can correctly generate and test new solutions, discovering an optimal solution on the basis of indirect feedback. It also establishes that ESPS can do structural credit assignment.

The second experiment extends the problem to a multi-step problem, where reinforcement is only received after the last step. This tests ESPS's ability to do temporal credit assignment.

5.2.1. The Single-Step Experiment

In the single-step experiment, ESPS's task is to converge on a randomly chosen n-bit word, called the target word. It does not know what the word is, and its only

feedback is the Hamming distance from its guess to the target. The situation is illustrated in figure 5.4. This is an instance of structural credit assignment because the feedback is indirect. Knowing the distance between a guess and the target does not give a direct indication of how the guess can be improved.

Before going on to discuss the algorithm, a word should be said about the problem. A programmer faced with solving the single-step problem and allowed to use any means at his disposal would not choose to solve the problem using ESPS. It is easy enough to devise an algorithm which would solve it in m steps, where m is the number of bits in the target word. The algorithm would start with a random word, and test each bit in turn. Bit k would be set to 1 and the feedback would be noted, then set to 0 and the feedback noted. It would be then set to the value for which it received the most feedback.

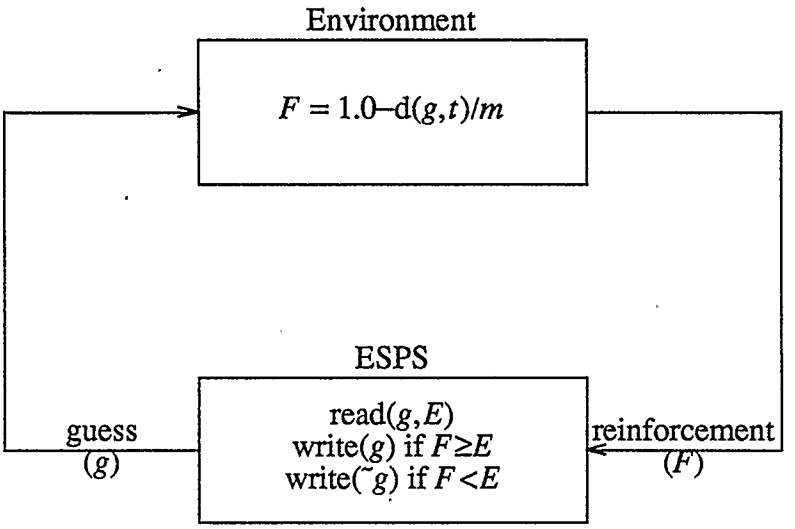


Figure 5.4. Single-Step Experiment.

Such a solution suffers from several drawbacks. First, it cannot generalise its results from one solution to other similar problems. The ability to generalise is a natural emergent property of SPS. Second, it works only on unimodal problems - those problems with no local minima. The algorithm works by making small changes in the solution and testing the effects of those changes. It is effectively doing a gradient descent. Were the problem multimodal, it could conceivably get trapped in a local minimum.

The reason for testing ESPS on a unimodal problem was to try the simplest problem possible which was an instance of reinforcement learning. Further experimentation will test its abilities on multimodal problems.

As shown in figure 5.4, the environment continually receives guesses (g) from ESPS, and produces feedback (F), according to the formula

$$F = 1.0 - \mathrm{d}(g,t)/m$$

where t is the target word, and $\mathrm{d}(g,t)$ is the Hamming distance between g and t. ESPS continually reads from memory, producing a guess and the expectation (E) for that guess. When it receives the feedback for that guess, it updates its memory, writing g when feedback exceeds expectations, writing $\tilde{g}$ when expectations exceed feedback.

In the test programs, the roles of Environment and ESPS are both played by the test program. That is, the environment and ESPS are not separate processes. The algorithm used in the single-step is illustrated in the pseudo-code below.

```
repeat
begin
 read(g, E)
 F := 1.0 - d(g,t)/m
 if F >= E then
  write(g)
 else
  write(~g)
end
```

Some tests used the binomial scheme to generate expectations, others used the linear scheme.

5.2.1.1. Single-Step Experiment Results

ESPS was tested on a series of values for m, ranging from 8 up to 128. All tests, with one exception, used a memory with 128-bit addresses, 10,000 locations, and a read/write circle of 49 bits. The exception occurred for $m = 128$, where, due to time considerations, an extremely small memory was used (1 actual location), with a read/write circle of 128 bits. The tests and their results are summarised in figure 5.5. For a given test, the number of trials to convergence is defined to be the first trial for

Word Size (m)	Words Per Address (p)	Test Runs	Average Trials To Convergence	Standard Deviation
8	5	10	>200	-
8	10	10	68.0	47.896
8	15	10	28.4	11.74
16	20	10	107.7	40.93
32	30	10	219.1	80.97
64	50	1	471	-
128	90	1	3626	-

Figure 5.5. Single-Step Experiment Results.

which it and all subsequent guesses are correct.

As can be seen from the data in figure 5.5, several tests were run, varying both the word size (m) and the number of words per address (p). For most configurations 10 runs were made, although once again due to time considerations, only 1 run was made for $m = 64$ and $m = 128$. No standard deviation was calculated for the first test ($m = 8$ and $p = 5$), as it never succeeded in converging (a limit of 200 was placed on the number of trials in a test run for $m = 8$).

There are three salient points to note about the data, which will be discussed in detail in the next section. First, the number of trials needed to converge grew as m increased (a trial consists of a guess, the reinforcement given to that guess, and the adjustment of expectation through storage of the guess or its inverse). Also, the number of words per address that were required grew also, from 10 for $m = 8$ to 90 for $m = 128$. Finally, for constant m, as p increased the number of trials required for convergence decreased, as did the standard deviation.

To give a feel for the behaviour of ESPS on the single-step test, figure 5.6 shows a graph of expectation (E), feedback (F) and distance ($d(g,t)$) for a run with $m = 128$. For the most part, ESPS steadily improves its performance, although at times its performance decreases, as it follows unproductive leads. The main point is that ESPS does not suddenly "stumble" upon the correct answer. This is perhaps not surprising in light of the fact that there are 2^{128} possible guesses, and only one is correct.

ESPS successfully completed the single-step experiment for m ranging from 8 to 128. When m was 128, it converged to the correct result in 3626 trials.

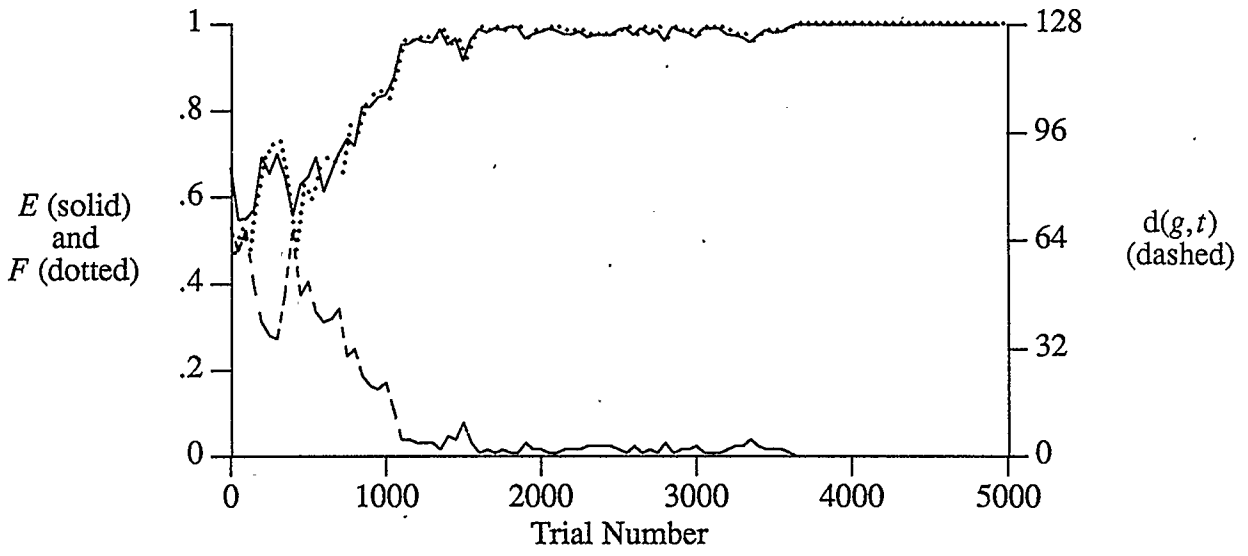


Figure 5.6. Single-Step Experiment Results for $n = 128$.

5.2.1.2. Discussion of Single-Step Experiment

The first point to be discussed is the increase in the trials to convergence as m increases. This is due to the increase in difficulty of the problem as m increases - the search space doubles for every increase in m of 1, so that the search space for $m = 128$ is 2^{64} times as large as for $m = 64$ for example. The results show an approximate doubling in convergence time for every doubling in m, with the exception of $m = 128$. The reasons for this exception are not clear but may be related to the number of words per address used in the $m = 128$ experiment. In the previous experiments, the number of words per address (p) was approximately the same as m. In the case of $m = 128$ though, p was quite a bit less. The tests run with $m = 8$ show the increase in performance as p increases, so the same could be expected if p was increased for $m = 128$. The effects of changing the number of words per address (p)

are discussed next.

To understand why the number of words per address must increase as m increases, the concepts of *correct* and *incorrect* decisions must be defined. These concepts are defined relative to a single, arbitrarily chosen bit in the guess. A *decision* is defined to be the choice of storing either the guess or its inverse. In terms of a single bit, then, a decision is the choice of storing either the bit or its inverse. The decision is correct when the bit stored is the correct value (equal to the target). The decision is incorrect when the bit stored is the wrong value (the inverse of the target).

The list of words at an address contains a record of the last p decisions made by the learning system involving that address, where p equals the number of words per address. If the number of correct decisions exceeds the number of incorrect decisions, then the probability of generating a correct guess is greater than 0.5 (assuming that the f function is the binomial or linear function). Furthermore, correct guesses are more likely to be stored as is (i.e., correctly), since a correct guess for a given bit will increase the chance that the entire guess is close to the target. Similarly, if the number of incorrect decisions is greater, then the probability of generating a correct guess is less than 0.5. Incorrect guesses increase the likelihood that the inverse will be stored (i.e., the correct decision is made), since an incorrect guess for a given bit will decrease the chance that the entire guess is close to the target. ESPS thus works to increase the population of correct guesses among the list of the last p decisions.

The reason why the number of words per address needed to be increased is due to the decrease in the reliability of the feedback for a given bit as m increases. That is, as m increases, the influence a single bit has on the the guess decreases, and therefore the influence of that bit on the feedback also decreases. The feedback received is thus less reliable as an indicator of whether that particular bit was correct or not. Be-

cause of this, correct decisions are less likely to be made. More decisions must be made and recorded before it becomes clear which is the correct value for a given bit. Recording more decisions means increasing the number of words per address.

The final point to discuss concerns the increase in performance as p increases for constant m. Three tests were run for $m = 8$, with p assuming values of 5, 10, and 15 (see figure 5.5). To test the change in performance, an hypothesis test was performed on the data.

Let us define test_0 to be the test where $p = 10$, and test_1 the test where $p = 15$. The hypothesis tested, H, is that there is an increase in performance from test_0 to test_1 that is statistically significant. That is, the mean of test_0 (μ_0) is less than the mean of test_1 (μ_1). The risk of type-I error we are willing to accept is $\alpha = 0.05$.

The hypothesis test to be run on the data requires that they be samples from a normally distributed population. However, because of the size of the samples, this cannot be established statistically. Instead, we will appeal to the central limit theorem, which tells us that a variable which is the sum of many statistically independent factors tends to be normally distributed. In the case of the operation of ESPS, the value of the variable (trials to convergence) is dependent on the the initial values given to each memory location, the target, and the particular pertubations of the guesses produced by the f function. I therefore assumed that the distribution of the number of trials to convergence is normal. Even if the distribution is not normal, it has been established that "violation of the assumption of normality has almost no practical consequences in using the t-test" (Glass and Hopkins, 1984, pg. 237).

An unpaired directional t-test was run on the data. The result of the test was that $\mu_0 < \mu_1$ with probability > 0.95. So the hypothesis is accepted, and we can conclude with reasonable certainty that the performance of test_1 is greater than that of test_0.

The explanation for the increase in performance lies in the size of p. The larger p is, the greater the base of old decisions on which to make the current decision. Assuming that correct decisions will be stored more frequently than incorrect decisions, a larger p will increase the chances that, for a given sequence of decisions, correct decisions will outnumber incorrect decisions.

These tests point out the shortcomings of storing a list of words at an actual location, instead of bit sums. When $m = 128$, 90 words were stored at each address, requiring $90 \times 128 = 11520$ bits per actual location. If bit sums were stored, it would only take $128 \times 8 = 1024$ bits, assuming 8 bits per bit sum (having 8 bits per bit sum would allow sums ranging from 0 to 255). Thus, bit sums use less storage space as soon as the number of words per address exceeds 8. Also, they are easier and quicker to work with. Writing would involve only incrementing and decrementing sums, rather than shifting every member of the list. Reading would no longer involve examining each bit of each word to create the bit sums, since those bit sums already exist. The current implementation was written assuming few words per address would be stored (around 4), and so lists of words rather than bit sums were used. Also, the theory of SPS was developed by Kanerva (1984) assuming word lists, so to ensure that the performance of the implementation kept as close as possible to that of the theory, word lists were used in the implementation.

5.2.1.3. Comparison to ASN

Tests were run on ESPS to compare its performance to the Associative Search Network (ASN), as described in Barto, Sutton and Brouwer (1981). Their experiment involved the presentation of two <*situation,action*> pairs (or more accurately, two <*identifier,target*> pairs). One identifier would be presented, and ASN would generate a guess. Feedback was the inner product of the guess and the target (output of

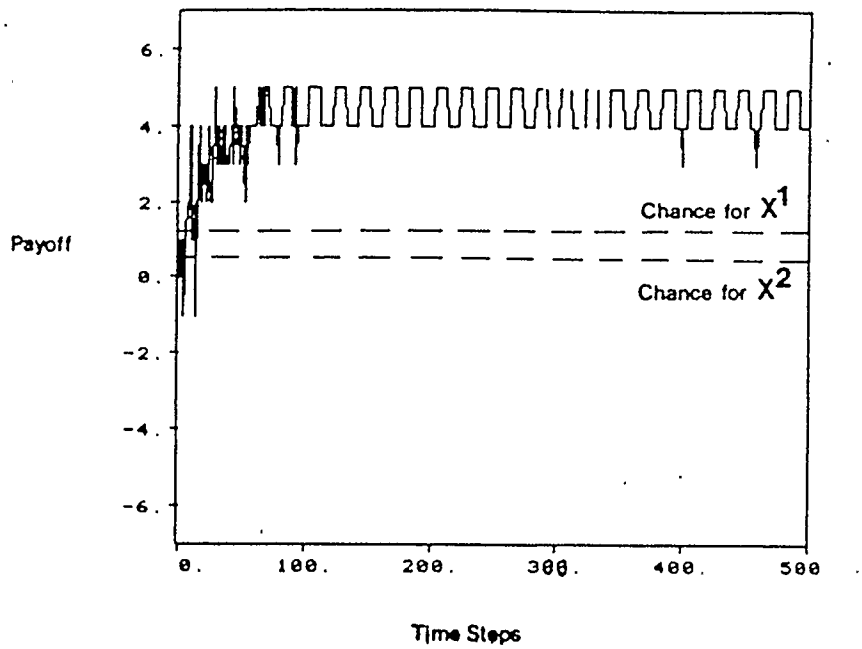


Figure 5.7. ASN Results (from Barto, Sutton, and Brouwer, 1981).

ASN is a vector of 1's and 0's, the target is a vector of 1's and -1's). This is essentially equivalent to Hamming distance. This step was repeated 10 times for one identifier, then 10 for the other, etc. Their experiment was designed to establish two results. First, that it is possible to derive the correct result using only indirect feedback. Second, that it could be done with an associative network. Their results are summarised in figure 5.7.

The values $\mathbf{X}^1$ and $\mathbf{X}^2$ are the target vectors. The line indicated "Chance for $\mathbf{X}^1$" is the expected payoff for a random guess of $\mathbf{X}^1$'s value. The size of $\mathbf{X}$ is 9.

A similar test was performed with ESPS, but with $m = 8$. That is, the target words were slightly smaller than that used for the ASN (the implementation of ESPS restricts words to be multiples of 8). The results of the test are shown in figure 5.8.

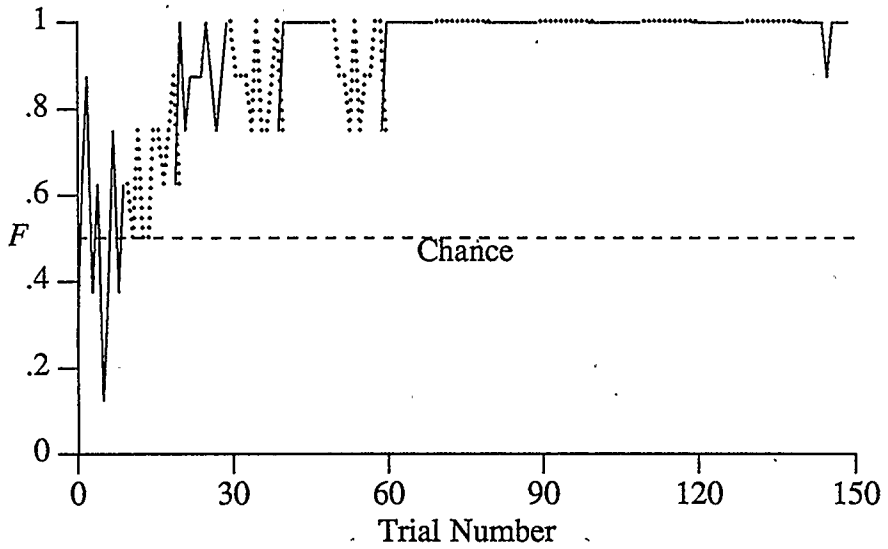


Figure 5.8. ESPS Results on ASN Test.

The expected feedback for a random guess is indicated by the dashed line labelled "Chance". Feedback for the first target word is indicated by the solid line; that for the second target word is indicated by the dotted line. Because ESPS converged faster than the ASN, only 150 trials are shown. Execution time was approximately 30 minutes when run on 3 Suns.

The experiments show that ESPS converges faster than the ASN. This is in part due to SPS's storage scheme. Given two arbitrary <*situation,action*> pairs, it is unlikely that they will have any actual locations in common because of the recoding stage between the inputs and the weights. The result of this is that learning one pair does not interfere with learning the other. In contrast, the ASN has no such recoding stage, and so there is interference between pairs. Because the pairs were linearly independent, there existed a set of weights that enabled them both to be eventually learned. However, learning was slowed by the interference between them.

5.2.1.4. Performance with Linear Rule

Tests were performed with ESPS using the linear f function. Performance of ESPS for $m = 8$ and 10 words per address is shown in figure 5.9.

Figure 5.9 shows that ESPS did not converge on the target word. This is not to say that guesses were completely random. After about 50 trials, feedback was consistently greater than 0.5 (the expected value if guesses were random), averaging about 0.7. Therefore, some learning was taking place. However, it failed to progress any further. It seems that the expectation generated by the linear rule is too pessimistic - this causes guesses to be wider-ranging than necessary. That is, the expectation associated with a guess is too low an estimate of the goodness of the guess. When the comparison is made to the feedback to decide whether to reward or punish the guess, bad guesses are rewarded unnecessarily. These bad guesses are written to

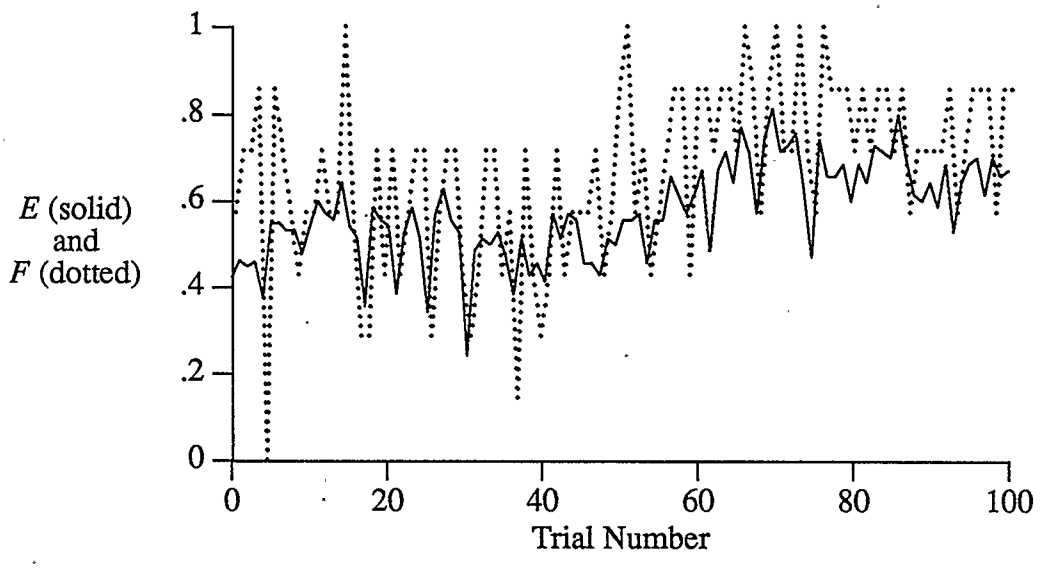


Figure 5.9. ESPS Performance with Linear f Function.

memory (a guess for which feedback exceeds expectation is written to memory as is). The effect is an increase in noise, negatively affecting the generation of the next guess. Tests with more words per address (50) met with the same problem.

5.2.2. The Multi-Step Experiment

The multi-step experiment was designed to test ESPS's ability to solve a task in which reinforcement is not given on each step.

In the multi-step experiment, the environment chooses a series of target words, t_1 to t_g. ESPS guesses each word in the sequence, with feedback coming at the end of the series of guesses. Feedback is calculated using the following formula:

$$f = \frac{\sum_{i=1}^{g} 1.0 - \frac{d(g_i, t_i)}{m}}{g}$$

That is, feedback is inversely proportional to the total distance between guesses and their corresponding target words.

The feedback scheme for the multi-step experiment is more complicated because only the last step receives environmental feedback. Steps which don't receive environmental feedback examine the expected feedback of the following step to update their own expectations. When their expectations exceed that predicted by the following step, then their expectations are lowered. When their expectations are exceeded by that predicted by the following step, then their own expectations are raised.

The pseudo-code for the multi-step experiment is rather difficult to illustrate, since the expectation for one step cannot be updated until the next one is read and its expectation calculated. It is more easily illustrated from the point of view of a bit location. The rule is:

$$w(t+1) = w(t)+greater(F(t+1),E(t))(2y(t)-1)$$

Where $w(t)$ is the value in the bit location at time t, $E(t)$ is the expectation at time t, and $greater(x,y)$ is 1 when $x \geq y$, -1 otherwise. $F(t+1)$ is a feedback signal that for the last step is the environmental feedback, and for all other steps is the expectation of the next step. Feedback for an action at time t is received at time $t+1$.

ESPS has not successfully completed the multi-step experiment. A test in which $m = 8$ is shown in figure 5.10. Each trial consisted of two steps. In the first step, an 8-bit guess was generated along with its expectation. The second step also generated an 8-bit guess and an expectation. This second expectation was the feedback for the first. The environmental feedback was the feedback used by the second step. Addresses were 128 bits long, there were 100 words per address, 10,000 actual locations, a read/write circle of 49 bits, and the binomial f function was used. Run time was 6 hours on 3 Suns. The graph shows the expectations generated on the first and

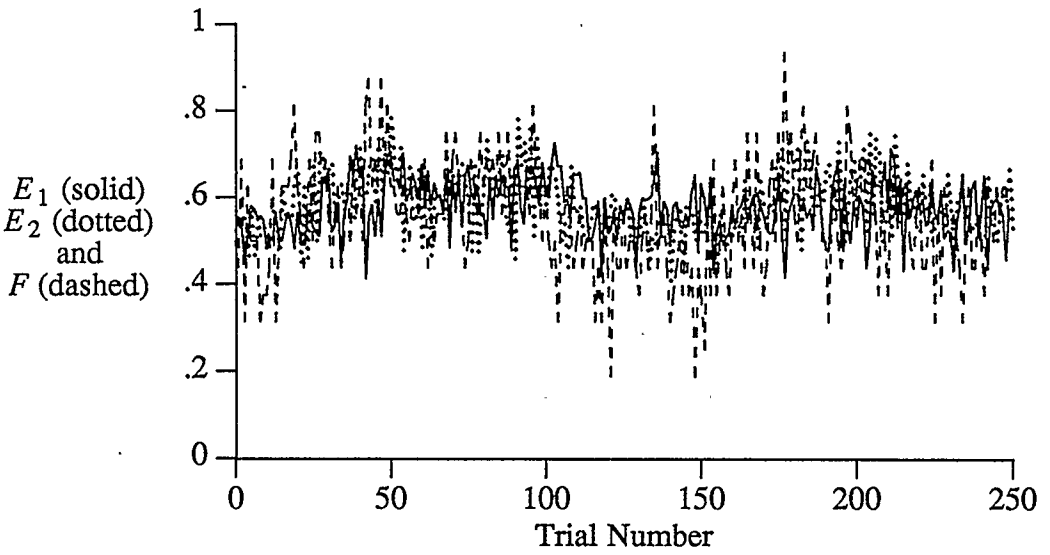


Figure 5.10. Multi-Step Experiment Results.

second steps (the solid and dotted lines, respectively), as well as the environmental feedback (the dashed line).

Figure 5.10 shows ESPS's inability to converge on the correct sequence of guesses. The problem lies with the widely-varying values for expectation generated on the second step. The first step relies on these values for adjusting its expectations. Such widely-varying values would send it confusing signals, making correct adjustments more difficult. Nevertheless, if correct decisions are more common than incorrect decisions, it should be possible to eventually converge on the correct answer.

Two tests were run in which correct and incorrect decisions were recorded. Both had the same memory configuration (8-bit words, 128-bit addresses, 10000 actual locations, 30 words per address, and a read/write circle size of 49 bits). The first test ran for 250 trials, the second for 1000. Both were two-step multi-step tests. Of particular interest is the number of correct and incorrect decisions made on the first step. In the case of the 250 trial run, 121 correct decisions were made, while 129 incorrect decisions were made. In the case of the 1000 trial run, 504 correct decisions were made, while 496 incorrect decisions were made.

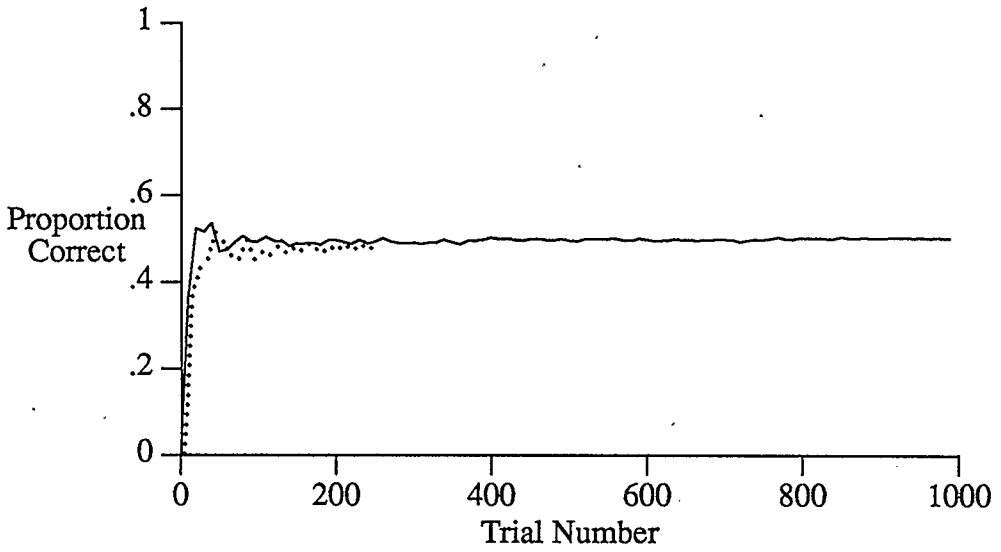
The reasons why the second test had more correct decisions than incorrect decision on the first step are unknown; it is possible that, given enough trials, correct decisions will always outnumber incorrect decisions. It is also possible that the fact that correct decisions outnumbered incorrect decisions on the second test was a chance event, and that correct decisions will in general not outnumber incorrect decisions. For the moment, assume that correct decisions *will* eventually always outnumber incorrect decisions, and that the proportion of correct decisions is very close to 0.5, as is the case in the second test.

Recall that in the discussion on the single-step experiment, it was stated that the list of words at an address contains a record of the last p decisions made by the learning system, where p equals the number of words per address. It was argued that as n increased, the reliability of the feedback decreased, meaning that correct decisions are less likely to be made. Therefore, as the proportion of correct decisions approaches 0.5, p must be made larger, large enough to contain a statistically significant sample of decisions. In the case of the second run of the multi-step experiment, the proportion is 0.504, too close to 0.5 for 30 words per address to be able to contain a statistically significant sample - it is quite possible that a group of 30 consecutive decisions will contain more incorrect than correct decisions. If 250 words per address were stored, we could expect about 126 correct and 124 incorrect decisions to be made. This would bias the output, perhaps enough to make convergence to the correct result possible. Unfortunately, due to the method used to implement ESPS, storing 250 words per address results in unmanageably large processes. Testing this hypothesis would require a reimplementation with bit sums rather than word lists.

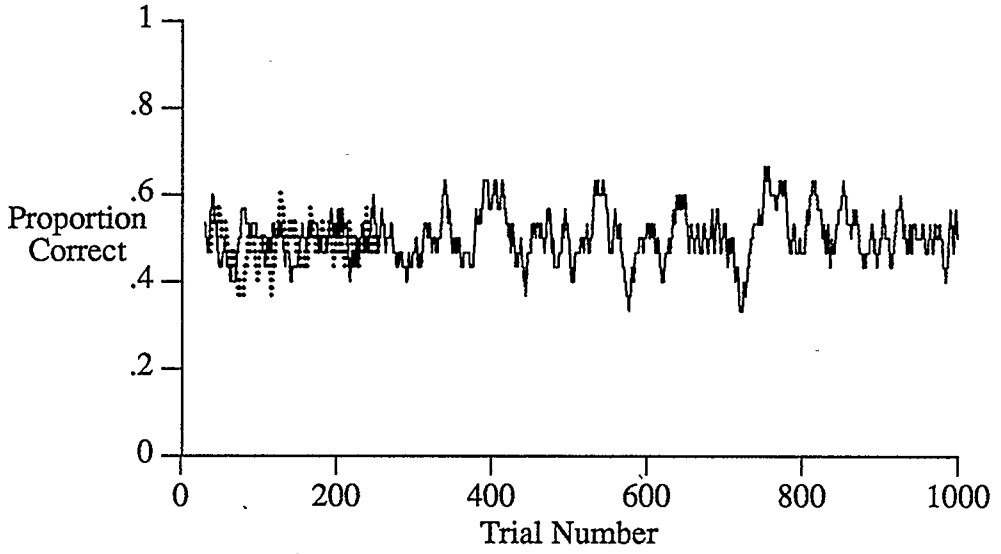
Figure 5.11 illustrates the results of the tests. Figure 5.11(a) shows the cumulative proportion of correct decisions at each trial. The first test (250 trials) is plotted with the dotted line, the second (1000 trials) with the solid line. Figure 5.11(b) shows the proportion of correct guesses over the last 30 guesses at each trial. Since each actual location contains 30 words, this gives an indication of the sample ESPS worked with to generate the next guess.

5.3. Conclusions

To summarise the results, ESPS has successfully solved the single-step learning problem, thus demonstrating its ability to generate new solutions and do structural credit assignment. Its performance compares favourably to that of another reinforce-



(a) Cumulative Proportion of Correct Guesses.



(b) Proportion Correct Over Previous Thirty Guesses.

Figure 5.11. Correct/Incorrect Test Results.

ment learning system, the Associative Search Network (Barto, Sutton, and Brouwer, 1981). ESPS has not solved the multi-step learning problem, and so has not demonstrated the ability to do temporal credit assignment.

The reasons for ESPS failure to solve the multi-step problem are not clear. ESPS is lacking a theory which can explain why it performs as it does. At the moment, adjustments such as changing the number of words per address are made on the basis of the effects of past adjustments that were found to work. The reasons why the adjustments work are unknown - all that is known is that they cause the intended results.

A theory of learning in ESPS could answer the following questions:

(1) What is the effect of different *f* functions? Why does the binomial function work while the linear one does not?

(2) Is learning in ESPS general, or does it just work for this particular combination of problem and *f* function? For example, can it cope with multimodal problems?

(3) Should the mechanisms of expectation and generation be separated, as with ASE/ACE (Barto, Sutton, and Anderson, 1983) and Holland's Classifier system (Holland, 1986)? Having them combined as in ESPS makes the range of values taken by reinforcement important. If reinforcement never reaches a high enough value, expectations will always be low, and thus generated solutions will vary considerably from step to step.

(4) Should expectations be calculated differently? The failure to solve the multi-step problem is due in part to the wide variations in the value of expectation for the second step, thus providing confusing feedback for the first step. The variations in expectation are caused by the manner in which it is calculated - the ex-

pectation for a word depends on the probable value of each bit and its actual value. It may be that expectations should depend only on the probable value of each bit.

CHAPTER 6

Discussion

6.1. Summary

Chapter 1 introduced reinforcement learning problems, which are those problems characterised by an environment which offers indirect and infrequent reinforcement. Two types of learning systems were introduced: supervised and reinforcement systems. Supervised systems require an environment which offers direct and frequent reinforcement. In effect, the environment has to know how to solve the problem. Reinforcement systems do not require an "intelligent" environment, needing only a rating of their performance rather than explicit solutions.

Chapter 2 surveyed supervised and reinforcement learning systems, and established that supervised systems, while capable of memorising <*situation*,*action*> pairs and performing generalisation, cannot solve reinforcement problems. Chapter 2 argued that reinforcement learning systems must overcome the problems of credit assignment and solution generation, and showed how each of the reinforcement systems covered did so.

Chapter 3 discussed SPS, a supervised system which is a model of the cerebellum. Its properties and its realisation with neuron-like components was discussed. Chapter 4 presented ESPS, an extension of SPS which can solve reinforcement problems. That is, it can generate new solutions and perform credit assignment. The ability to generate solutions was added by making reading non-deterministic. Credit assignment was added by introducing the idea of *expectation* and by extending the notion of feedback to include the expectations of succeeding steps. Physiological

justification for these changes was presented. Non-deterministic reading was justified by assuming that neurons in the cerebellum are imperfect (they were assumed to work perfectly in SPS). The extended notion of feedback required redefining the role of climbing fibres, and by postulating that the expectations generated during reading could be broadcast by interneurons, perhaps by a path that left the cerebellum and returned via the climbing fibres.

Chapter 5 presented the results of experimentation with ESPS. ESPS was tested on two experiments, one designed to test its ability to generate new solutions and do structural credit assignment (the single-step experiment), the other designed to test its ability to do temporal credit assignment (the multi-step experiment). ESPS solve the single-step problem, but did not solve the multi-step problem. In discussing its performance on the single-step problem, the concepts of *correct* and *incorrect* decisions were introduced. These concepts were used to explain why an increased number of words per address were required as m (the size of the target word) increased. They were also used in explaining ESPS's failure to solve the multi-step problem. It was postulated that ESPS *could* solve the multi-step problem if the number of words per address were increased to approximately 250.

In addition, ESPS was tested on two f functions (binomial and linear), and was found to be sensitive to the particular f function used. It could solve problems using the binomial f function, but not with the linear function. ESPS was also tested on an experiment similar to one performed with the Associative Search Network (Barto, Sutton, and Brouwer, 1981). Its performance was found to be comparable to ASN's.

6.2. Conclusions

Solving reinforcement problems is important, and difficult. Important, because it breaks the reliance on an environment which is "smarter" than the learning system.

Difficult, because it requires translating a "low quality" feedback signal which only indirectly specifies correct behaviour into a set of rules producing the correct behaviour.

A reinforcement learning system must be able to correctly assign credit and generate new solutions. Because reinforcement is infrequent, each step in a solution is not rated. Therefore, the reinforcement learning system needs a scheme which can correctly assign ratings, or credit, to each step of the problem. Because the reinforcement is indirect, providing only a rating of the overall behaviour of the system, it cannot be used directly to correct the solutions generated by the learning system. The learning system must be able to generate new solutions and judge the effects of those solutions.

ESPS is interesting not only because it can solve a reinforcement learning problem, but also because it is based on a model of the cerebellum, SPS. Because ESPS is based on a model of the cerebellum, and because the changes made to SPS were physiologically plausible, it suggests that the cerebellum may be capable of solving reinforcement learning problems.

The conclusions that can be drawn from this thesis are:

(1) *The credit assignment problem has structural and temporal forms.* When a solution is produced as the combined results of several portions of a system, the credit, or rating, given to the solution must be properly assigned to each portion. This is a structural credit assignment problem. When a solution is composed of several steps, the credit given to the solution must be properly assigned to each step. This is a temporal credit assignment problem.

(2) *It is possible for ESPS to solve a reinforcement learning problem.* Its success in the single-step problem shows that ESPS can solve the structural credit assign-

ment problem, and its performance was comparable to that of ASN (Barto, Sutton, and Brouwer, 1981). It did not succeed in solving the multi-step problem, which is a problem of temporal credit assignment, but there is evidence to suggest that this failure is due to storing too few words at each actual location, rather than a some qualitative shortcoming.

(3) *The ability of ESPS to solve a problem is dependent on the particular f function used.* The experimental results showed that, while able to solve the single-step problem using the binomial f function, f_B, ESPS was unable to solve the problem under the same conditions using the linear f function, f_L. The results are not conclusive, however, and it would be premature to state that ESPS could *never* solve the single-step problem using f_L. We *can* conclude that the performance of ESPS is sensitive to the particular function used.

(4) *The ability of ESPS to solve a problem is dependent on the number of words stored per address.* The experimental results showed that a certain minimum number of words per address is required before ESPS can converge on the correct result in the single-step problem. Furthermore, the results of the multi-step tests suggest that ESPS could solve the multi-step test if the number of words per address were increased to a value of approximately 250. Unfortunately, the current implementation makes testing that hypothesis impractical.

6.3. Future Work

Future work includes a reimplementation of ESPS using bit sums at actual locations instead of lists of words. With this reimplementation, it would be possible to test the hypothesis that the multi-step problem could be solved by having the equivalent of 250 words per address.

At the present, ESPS can only solve a unimodal, single-step reinforcement learning problem. To cope with infrequent reinforcement it must be able to solve the multi-step problem. Furthermore, both the single- and multi-step problems are unimodal. Dealing with multimodal problems is significant, as can be seen by the amount of effort devoted to solving such problems in supervised systems. ESPS's use of a probabilistic function to compute output means theoretically that any local minimum can be escaped (unless the probabilities are 1 or 0). However, the time to escape these minima and find the global minimum may be impractically large.

At the moment, questions regarding the rate of convergence can only be answered through experimentation. Developing a mathematical model of ESPS would help in answering this question, as well as others concerning its ability to solve multimodal problems, and the effects of various *f* functions.

The tests devised so far do not test many of the capabilities of ESPS inherited from SPS. SPS has the ability to generalise, producing actions for situations which have not been previously seen. The actions produced are a kind of average of similar, previously seen, situations. This is an important capability in problem domains where performing similar actions in similar situations is a reasonable strategy. Neither the single- and multi-step problem test this capability.

Finally, for ESPS to be applied to real-world problems, values from the world must be translated into the *n*-bit words used by ESPS. Real-world values are often real-value quantities, such as joint positions and velocities, or perhaps vectors of reals, such as an image bitmap. Each value will have a measure of similarity defined for it. The translation process is difficult because it must maintain the similarities that existed, in spite of these values being converted into *n*-bit vectors in which Hamming distance is the measure of similarity. Some preliminary work has been done in

this area (Schack, 1986).

References

Ackley, D.H., Hinton, G.E., and Sejnowski, T.J. (1985). A learning algorithm for Boltzmann Machines. *Cognitive Science, 9,* 147-169.

Albus, J. (1971). A theory of cerebellar function. *Mathematical Biosciences, 10,* 25-61.

Albus, J. (1975). A new approach to manipulator control: the cerebellar model articulation controller (CMAC). *Journal of Dynamic Systems, Measurement, and Control, 97,* 220-227.

Anderson, J.A. (1983). Cognitive and psychological computation with neural models. *IEEE Transactions on Systems, Man, and Cybernetics,* smc-13, 799-815.

Andreae, J.H. (1972). *Man-Machine Studies.* Progress Report no. UC-DSE/1(1972) to the Defence Scientific Establishment. Department of Electrical and Electronic Engineering, University of Canterbury, Christchurch, New Zealand. (This report is also available from the N.T.I.S., 5285 Port Royal Rd, Springfield, Virginia 22161).

Andreae, J.H. (1977). *Thinking with the Teachable Machine.* Academic Press.

Andreae, J.H., and MacDonald, B.A. (1987). Expert control for a robot body. Research Report no. 87/286/34, Department of Computer Science, University of Calgary.

Arshavsky, Y.I., Gelfand, I.M., and Orlovsky, G.N. (1986). *Cerebellum and Rhythmical Movements*. Springer-Verlag: Berlin, Heidelberg.

Barto, A.G., Sutton, R.S., and Brouwer, P.S. (1981). Associative Search Network: A reinforcement learning associative memory. *Biological Cybernetics,* 40, 201-211.

Barto, A.G., Sutton, R.S., and Anderson, C.W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics,* smc-13, 834-846.

Blomfield, S., and Marr, D. (1970). How the cerebellum may be used. *Nature,* 227, 1224-1228.

Chou, P.A. (1987), The capacity of the Kanerva associative memory is exponential. To be published in Collected papers of IEEE conference of Neural Information Processing Systems, Denver, Colorado, November, American Institute of Physics.

Cohen, P.R., and Feigenbaum, E.A. (1982). *The Handbook of Artificial Intelligence, Volume 3*. Los Altos, California: William Kaufmann, Inc.

Glass, G.V., and Hopkins, K.D. (1984). *Statistical Methods in Education and Psychology*. Second edition, Prentice-Hall: Englewood Cliffs, New Jersey.

Holland, J.H. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (Editors), *Machine Learning,* (volume II, pp. 593-623). Los Altos, California: Morgan Kaufmann.

Hopfield, J.J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences, USA, 79,* 2554-2558.

Howard, R.A. (1966). *Dynamic Programming and Markov Processes.* MIT Press.

Jordon, M.I. (1986). An introduction to linear algebra in parallel distributed processing. In *Parallel Distributed Processing: Explorations in the Microstructure of* Cognition, Vol. 1, D.E. Rumelhart, J.L. McClelland, (Editors), Bradford: Cambridge, MA.

Kanerva, P. (1984). Self-propagating search: A unified theory of memory, (doctoral dissertation, Stanford University), *University Microfilms,* 1984.

Keeler, J.D. (1987). Information capacity of outer-product neural networks. *Physics Letters A,* Volume 124, number 1,2, 53-58.

Kirkpatrick, S., Gelatt, C.D. Jr., and Vecchi, M.P. (1983). Optimization by simulated annealing. *Science, 220,* 671-680.

Kohonen, T. (1984). *Self Organization and Associative Memory.* Berlin: Springer-Verlag.

Llinás, R.R. (1975). The cortex of the cerebellum. *Scientific American, 232(1),* 56-71.

Marr, D. (1969). A theory of cerebellar cortex. *The Journal of Physiology, 202,* 437-470.

Miller, W. Thomas III, Glanz, F.H., Kraft, L.G. III (1987). Application of a general learning algorithm to the control of robotic manipulators. *The International Journal of Robotics Research,* Volume 6, Number 2, 84-98.

Minsky, M., and Papert, S. (1969). *Perceptrons*. Cambridge, MA: MIT Press.

Narendra, K.S., and Thathachar, M.A.L. (1974). Learning automata - a survey. *IEEE Transactions on Systems, Man, and Cybernetics, 4,* 323-334.

Nash, L.K. (1974). *Elements of Statistical Thermodynamics,* (second edition), Addison-Wesley: Reading, MA.

Pellionisz, A.J. (1984). David Marr: A theory of the cerebellar cortex. In *Brain Theory,* G. Palm, A. Aertsen, (Editors), Springer-Verlag: Berlin, Heidelberg.

Rosenblatt, F. (1962). *Principles of Neurodynamics*. Washington, D.C.: Spartan.

Rumelhart, D.E., Hinton, G.E., and Williams, R.J. (1986). Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of* Cognition, Vol. 1, D.E. Rumelhart, J.L. McClelland, (Editors), Bradford: Cambridge, MA.

Samuel, A.L. (1963). Some studies in machine learning using the game of checkers. In Feigenbaum, E.A. and Feldman, J. (Editors) *Computers and Thought,* (pp. 71-105). New York: McGraw-Hill.

Schack, B. (1986). *Self-Propagating Search: A Study.* (final report for CPSC 601.69, University of Calgary).

Sejnowski, T. (1986). Open questions about computations in cerebral cortex. In *Parallel Distributed Processing: Explorations in the Microstructure of* Cognition, Vol. 2, J.L. McClelland, D.E. Rumelhart, (Editors), Bradford: Cambridge, MA.

Unger, B.W., Dewar, A., Cleary, J., and Birtwistle, G.M. (1986). A distributed software prototyping and simulation environment: Jade. *Proceedings of the Conference on Intelligent Simulation Environments, 17 (1), San Diego,* 63-71, SCS Simulation Series.

Widrow, B., Gupta, N.K., and Maitra, S. (1973). Punish/reward: Learning with a critic in adaptive threshold systems. *IEEE Transactions on Systems, Man, and Cybernetics, 3,* 455-465.