Learning and Problem Solving with Multilayer Connectionist Systems

A Dissertation Presented

By

Charles William Anderson

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

Doctor Of Philosophy

September 1986

Department of Computer and Information Science

This research was supported by the Air Force Office of Scientific Research and the Avionics Laboratory (Air Force Wright Aeronautical Laboratories) through contracts F33615-77-C-1191, F33615-80-C-1088, and F33615-83-C-1078.

Dedicated to

Stacey and Joseph

Acknowledgements

Research in areas where there are many possible paths to follow requires a keen eye for crucial issues. The study of learning systems is such an area. Through the years of working with Andy Barto and Rich Sutton, I have observed many instances of "fluff cutting" and the exposure of basic issues. I thank both Andy and Rich for the insights that have rubbed off on me. I also thank Andy for opening up an infinite world of perspectives on learning, ranging from engineering principles to neural processing theories. I thank Rich for showing me the most important step in doing "science"—simplify your questions by isolating the issues.

Several people contributed to the readability of this dissertation. Andy spent much time carefully reading several drafts. Through his efforts the clarity is much improved. I thank Paul Utgoff, Michael Arbib, and Bill Kilmer for reading drafts of this dissertation and providing valuable criticisms. Paul provided a non-connectionist perspective that widened my view considerably. He never hesitated to work out differences in terms and methodologies that have been developed through research with connectionist vs. symbolic representations. I thank Stephen Judd, Brian Pinette, Jonathan Bachrach, and Robbie Jacobs for commenting on an early draft and for many interesting discussions.

I thank Andy, Rich, Harry Klopf, Michael Arbib, Nico Spinelli, Bill Kilmer, and the AFOSR for starting and maintaining the research project that supported the work reported in this dissertation. I thank Susan Parker for the skill with which she administered the project. And I thank the COINS Department at UMass and the RCF Staff for the maintenance of the research computing environment. Much of the computer graphics software used to generate figures of this dissertation is based on graphics tools provided by Rich Sutton and Andy Cromarty.

Most importantly, I thank Stacey and Joseph for always being there to lift my spirits while I pursued distant milestones and to share my excitement upon reaching them. Their faith and confidence helped me maintain a proper perspective.

Abstract

Learning and Problem Solving
with Multilayer Connectionist Systems

September 1986

Charles William Anderson

B.S., University of Nebraska

M.S., University of Massachusetts

Ph.D., University of Massachusetts

Directed by: Professor Andrew G. Barto

The difficulties of learning in multilayered networks of computational units has limited the use of connectionist systems in complex domains. This dissertation elucidates the issues of learning in a network's *hidden* units, and reviews methods for addressing these issues that have been developed through the years. Issues of learning in hidden units are shown to be analogous to learning issues for multilayer systems employing symbolic representations.

Comparisons of a number of algorithms for learning in hidden units are made by applying them in a consistent manner to several tasks. Recently developed algorithms, including Rumelhart, et al.'s, error back-propagation algorithm and Barto, et al.'s, reinforcement-learning algorithms, learn the solutions to the tasks much more successfully than methods of the past. A novel algorithm is examined that combines aspects of reinforcement learning and a *data-directed* search for useful weights, and is shown to out perform reinforcement-learning algorithms.

A connectionist framework for the learning of strategies is described which combines the error back-propagation algorithm for learning in hidden units with Sutton's AHC algorithm to learn evaluation functions and with a reinforcement-learning algorithm to learn search heuristics. The generality of this hybrid system is demonstrated through successful applications to a numerical, pole-balancing task and to the Tower of Hanoi puzzle. Features developed by the hidden units in solving these tasks are analyzed. Comparisons with other approaches to each task are made.

Contents

Dedications 3 Acknowledgements 4 Abstract 5

1 Introduction **9**
1.1 Connectionist Systems . . . 10
1.2 Credit Assignment in Multilayer Systems . . . 14
1.3 Reinforcement Learning . . . 15
1.4 Problem Solving . . . 16
1.5 Research Objective and Method . . . 17
1.6 Reader's Guide . . . 18

2 New Features and the Facilitation of Learning **19**
2.1 The Problem of Missing Features . . . 20
2.1.1 Missing Terms in LEX . . . 20
2.1.2 The Multiplexer Task . . . 23
2.1.3 The Perceptron Learning Algorithm . . . 24
2.1.4 Performance Measures . . . 24
2.1.5 Original Representation . . . 25
2.1.6 An Ideal Representation . . . 25
2.1.7 A Representation Resulting in No Generalization . . . 26
2.1.8 New Features Added to Original Representation . . . 27
2.1.9 Results . . . 27
2.2 New Features That Add Beneficial Generalization . . . 29
2.2.1 The Input-Cluster Task . . . 30
2.2.2 Original Representation . . . 30
2.2.3 New Features—Basis Vectors as Class Labels . . . 30
2.2.4 New Features—Class Labels with Generalization . . . 31
2.2.5 Results . . . 31
2.3 New Output Features—Correlations Among Output Components . . . 33
2.3.1 The Output-Vector Task . . . 34
2.3.2 Changes to Performance Measures . . . 35
2.3.3 Original Representation . . . 35
2.3.4 New Features That Correlate Output Components . . . 35
2.3.5 Results . . . 37
2.4 New Intermediate Concepts in Production Systems . . . 38
2.4.1 Taxonomy Points . . . 39
2.4.2 Switchover Points . . . 39
2.5 Summary . . . 40

3 Review of Learning Methods for Hidden Units **41**
3.1 The Connectionist Learning Problem . . . 41
3.2 Direct Search . . . 42
3.3 Assignment of Credit . . . 43
3.3.1 Exact Gradient Methods . . . 43
3.3.2 Approximate Gradient Methods . . . 45
3.3.3 Minimal Change . . . 48
3.3.4 Worth . . . 51
3.4 Modification . . . 53
3.4.1 Fraction of Gradient . . . 53
3.4.2 Minimal Change . . . 53
3.4.3 Generate New Units . . . 54
3.5 Outline . . . 55
3.5.1 Direct Search . . . 55
3.5.2 Gradient Methods . . . 55
3.5.3 Minimal Change . . . 58
3.5.4 Worth . . . 60

4 Comparison of Methods for Learning Missing Features **63**
4.1 Direct-Search Algorithms . . . 65
4.1.1 Unguided Random Search . . . 65
4.1.2 Guided Random Search . . . 67
4.1.3 Polytope Algorithm . . . 69
4.2 Error Back-Propagation Algorithms . . . 72
4.2.1 Rosenblatt . . . 72
4.2.2 Rumelhart, Hinton, and Williams . . . 73
4.3 Reinforcement Learning . . . 77
4.3.1 Associative Search with Reinforcement Prediction . . . 77
4.3.2 Associative Reward-Penalty . . . 78
4.3.3 Local Reinforcement . . . 81
4.3.4 Penalty Prediction . . . 82
4.4 Summary . . . 85

5 Strategy Learning with Multilayer Connectionist Systems **91**
5.1 Strategy Learning Behavior of the Algorithms . . . 91
5.1.1 Initial Search Strategy . . . 91
5.1.2 Credit Assignment . . . 92
5.1.3 Modification . . . 93
5.2 Connectionist Algorithms for Strategy Learning . . . 94
5.2.1 Output Functions . . . 95
5.2.2 Learning Algorithms . . . 96
5.2.3 Parameters . . . 99

6 Learning a Solution to a Numerical Control Task **100**
6.1 The Pole-Balancing Task . . . 100
6.2 Control-Engineering Approach . . . 102
6.3 Our Approach . . . 102
6.4 Experiments . . . 103

6.4.1 Simulation . . . 103
6.4.2 Desired Functions . . . 103
6.4.3 Interaction between Learning System and Cart-Pole Simulation . . . 105
6.4.4 Results of One-Layer Experiments . . . 105
6.4.5 Results of Two-Layer Experiments . . . 108
6.5 Transfer to Similar Tasks . . . 114
6.6 Conclusion . . . 114

7 Learning the Solution to a Puzzle 117
7.1 The Tower of Hanoi Puzzle . . . 117
7.2 Experiments . . . 120
7.2.1 Representation of States and Actions . . . 120
7.2.2 Credit Assignment . . . 120
7.2.3 Interaction between Learning System and Puzzle . . . 121
7.2.4 Results of One-Layer Experiments . . . 122
7.2.5 Results of Two-Layer Experiments . . . 125
7.2.6 The Evaluation Function and New Features . . . 128
7.2.7 The Action Function . . . 130
7.3 Transfer of Learning . . . 131
7.4 Conclusion . . . 133

8 Summary and Future Work 134
8.1 Comparison of Methods for Learning in Hidden Units . . . 134
8.2 Strategy Learning with Multilayer Connectionist Systems . . . 135
8.2.1 Pole Balancing . . . 136
8.2.2 Tower of Hanoi . . . 136
8.3 Future Work . . . 138
8.4 Applications . . . 138

A Results from One-Layer Network Experiments of Chapter II 140

B Derivation of $P(a_j = 1)$ for Tower of Hanoi Experiments 144
B.1 Two Legal Actions . . . 144
B.2 Three Legal Actions . . . 145

C Weights for Solutions of Strategy Learning Tasks 147
C.1 Pole-Balancing Task . . . 147
C.2 Tower of Hanoi Task . . . 147

Bibliography 151

Chapter 1

Introduction

Connectionist systems embody a framework for decision-making based on an active form of knowledge representation. They are composed of simple computational units interconnected by pathways that transmit numerically-valued signals rather than complicated symbolic messages. The connectionist framework stems both from psychological theories of how the brain represents associations among concepts and from the modeling of neural networks. Current methods and applications of connectionist systems constitute a paradigm at a level between symbolic and neural representations. Although connectionist and symbolic representations are not on the same level, they are subject to analogous problems concerning the development of a representation. *Representation development* is the process whereby modifications are made to a representation by the addition, removal, or alteration of the representation's components, whether these components are symbolic terms or numerically-valued features.

Interest in connectionist systems has grown in recent years for several reasons. The inherent parallelism of connectionist systems can result in fast decision-making. Also, the use of connectionist systems as models of cognitive processes has met with some success. For example, Rumelhart and McClelland's (1986) connectionist system models the verb-tense learning behavior of children, a behavior often presented as an argument for the symbolic representation of explicit rules. A third reason for the growing interest in connectionist systems is the recent progress towards a solution to the problem of learning in multilayer connectionist systems, overcoming an obstacle that has been a major criticism of the connectionist paradigm supported by Minsky and Papert's (1969) analysis of the limitations of the perceptron (Rosenblatt, 1962).

The issues of learning in connectionist systems, however, are far from resolved. The work reported in this dissertation addresses three pressing learning issues:

1. relationships among various approaches to learning in multilayer connectionist systems,
2. the *structural* credit-assignment problem (defined below), and
3. learning in cases in which the desired output of the system is unknown.

The relationships among learning algorithms for multilayer connectionist systems are elucidated by reviewing them within a framework based on a categorization of methods for structurally assigning credit. Numerical learning algorithms not originally presented as connectionist learning algorithms are also discussed within this framework. Such a consistent review is much needed in a field where researchers come from diverse backgrounds, as is the case for connectionist learning research.

Some of the better-known learning algorithms for multilayer connectionist systems are studied by applying them to the multiplexer learning task (described in Chapters II and IV). In comparing the performances of different algorithms, consideration was given to details such as consistency in the training procedure, optimization of the algorithms' parameters, measures both for the performance level during learning and at the conclusion of a learning run, and statistical confidence

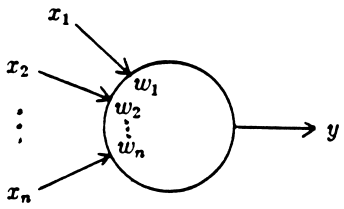


Figure 1.1: A Linear Threshold Unit

intervals for all data. Such careful comparisons rarely appear in the literature but are necessary for drawing significant conclusions.

The third issue listed above is discussed in terms of reinforcement-learning methods, as described later in this chapter. To date, research with reinforcement-learning methods has focused on single-layer learning systems (Barto and Anandan, 1985; Barto, Sutton, and Anderson, 1983; Barto, Sutton, and Brouwer, 1981; Barto and Sutton, 1981), though their potential use for learning in multilayer connectionist systems has been demonstrated in several small examples (Anderson, 1982; Barto, 1985; Barto, Anandan, and Anderson, 1986; Barto and Anderson, 1985; Barto, Anderson, and Sutton, 1982). In this thesis, reinforcement-learning methods are combined with a learning algorithm for multilayer systems to develop an example of a multilayer connectionist system for the learning of problem-solving strategies. This system is demonstrated on a pole-balancing control task and on a Tower of Hanoi puzzle.

1.1 Connectionist Systems

Connectionist systems generally consist of a collection of computational units, sometimes described as neuron-like in their input-output behavior. Each unit receives a number of input signals, or *input components*, whose numerical values constitute the unit's *input vector*, and the unit applies an *output function* to its input to generate output values. *Networks* of units are constructed by connecting the output of some units to the input of other, or the same, units. A network is said to interact with an *environment* by receiving a vector of numerical values from the environment and producing an output vector that acts upon the environment. Thus a unit's input components can originate either from the network's environment or from the output of another unit; a unit's output can be passed on to another unit or it can become a component of the network's output.

A unit's output function is parameterized by a vector of numerical weights, one weight for every input component. For a given network structure, it is the values of these weights that determines the input-output behavior of the network. A *learning algorithm*[1] for a connectionist system is a method for updating the values of the system's weights based on the performance of the system. For reviews of current connectionist research see Feldman and Ballard (1982), Hinton and Anderson (1981), McClelland and Rumelhart (1986), and Rumelhart and McClelland (1986).

The output functions performed by the units are usually one of a small set of functions. The most common function is the linear threshold function used by the pioneers of adaptive networks (e.g., Farley and Clark, 1954; McCulloch and Pitts, 1943; Rosenblatt, 1962; Widrow, 1962). A unit implementing this function is sketched in Figure 1.1, where $x_1, x_2, \ldots, x_n$ are the components of the unit's n-dimensional input vector, $w_1, w_2, \ldots, w_n$ are the unit's weights, and y, the unit's

[1] "Algorithm" is not used here in the strictest sense of Knuth's (1973) definition: finiteness is not assumed. Knuth suggests that such procedures be called "computational methods", but for brevity we will use "algorithm." Also, by using the label "learning algorithm" we do not imply the existence of a proof of convergence to a desirable outcome.

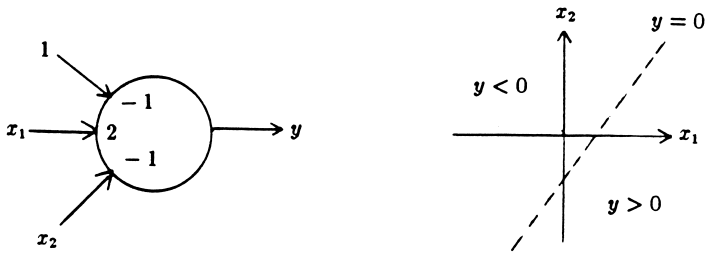


Figure 1.2: Discrimination of a Linear Threshold Unit

output, is defined as

$$y = \begin{cases} 1, & \text{if } \sum_{i=1}^{n} x_i \, w_i > 0; \\ 0, & \text{otherwise.} \end{cases}$$

The threshold of this unit is zero, but it can be considered to be parameterized if one of the input components is constant (so that the value of the corresponding weight becomes the negative of the threshold[2]). Such a unit can discriminate (i.e., produce different output values for) sets of input patterns that are linearly-separable (can be separated by an $n-1$ dimensional surface). For example, a unit receiving two input components discriminates input vectors by a line, as shown in Figure 1.2.

Other related output functions include the following (most are discussed further in later chapters). Hinton and Sejnowski (1983) and Barto, et al. (e.g., Barto, 1985; Barto, Sutton, 1981a; Barto, Sutton, and Brouwer, 1981), add noise to the weighted sum of a unit's inputs, effectively giving the unit a noisy threshold. The unit chooses one of the possible output values (0 and 1) with a probability determined by the unit's weights and input vector. Rumelhart, Hinton, and William's (1986) *semilinear* unit employs a "smoothed" version of the linear threshold function that is a continuous, differentiable function.

Another type of function is that of a *prototype* unit (Reilly, Cooper, and Elbaum, 1982), or template-matching unit (Uhr and Vossler, 1961), which produces the largest output values for input vectors with components that are identical to their corresponding weight values. Smaller output values are generated as the input vector becomes less similar to the weight vector. The weight vector forms a prototype for the concept encoded by the unit. A linear unit can be viewed as a prototype unit when its input components have values 0 or 1.

Many network architectures have been used for interconnecting units. We focus on networks having no cycles, i.e., no recurrent connections. Units only participate once in the computation of the system's output for a given input. The dynamical behavior of recurrently-connected networks complicates learning issues, but some approaches to learning are applicable to special cases (Ackley, Hinton, and Sejnowski, 1985; Rumelhart, Hinton, and Williams, 1986).

Figure 1.3a shows what is meant by the layers of a learning system in terms of an adaptation of the general model of learning systems described by Dietterich (Cohen and Feigenbaum, 1982, Ch. 14). A two-layer system is shown in which both layers receive input from the environment, and the last layer, or *output layer*, generates the output of the system which affects the environment. The outputs of the first layer become additional input components to the second, or output, layer, thus forming *new features* that extend the input representation utilized by the output layer.

A two-layer connectionist network is drawn in Figure 1.3b. The correspondences between Dietterich's model and the connectionist model are as follows: the performance element of a layer is the set of output functions for the units of that layer; the knowledge base is the set of weight values; and the learning element is the scheme for updating the weight values. We will assume that the *critic* uses some performance standard to assign errors to the output layer or a scalar evaluation

[2]Using a constant input is a standard technique for implementing a variable threshold. See Nilsson (1965) for further details.

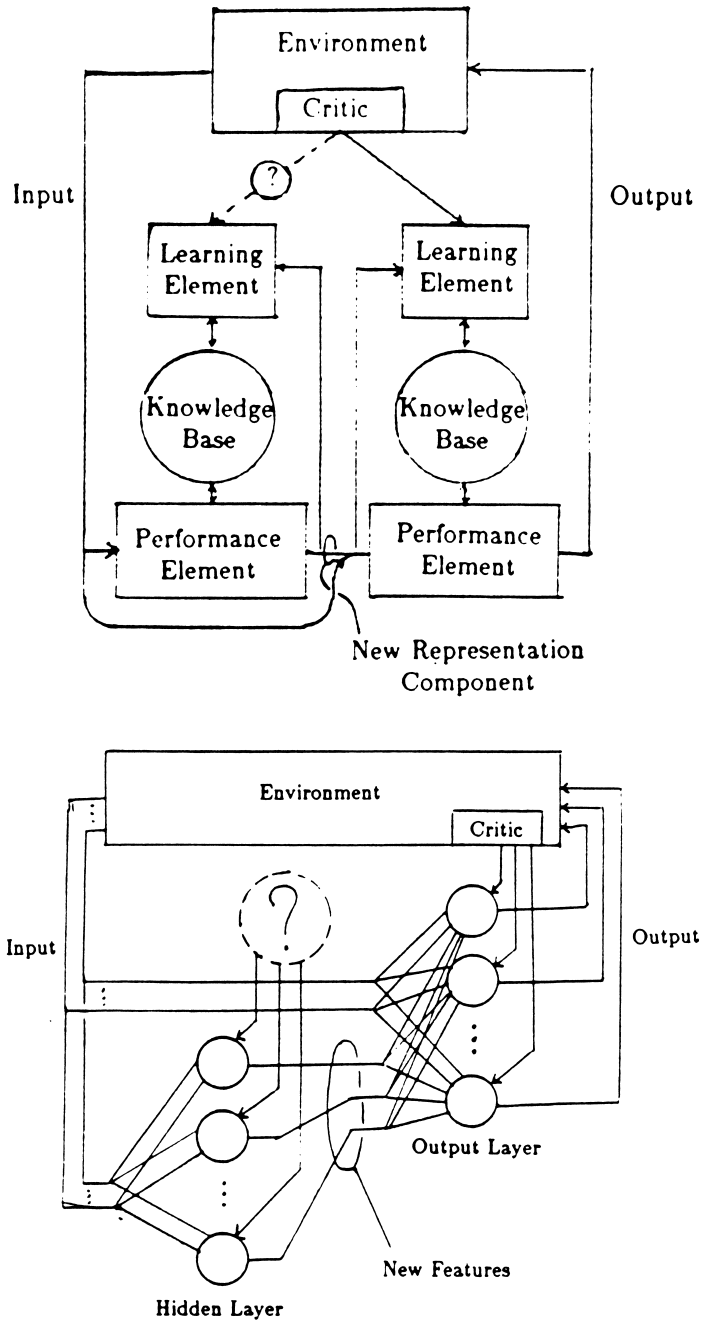


Figure 1.3: Model of a Two-Layer Learning System

when errors cannot be determined. This formulation excludes *unsupervised* learning methods which do not depend on the network's output and its relation to the goal of the task. This thesis focuses on tasks for which the learning of a solution requires the closed-loop interaction between system and environment that is absent from unsupervised learning methods (see Barto and Sutton, 1981b, for further discussion on categories of learning), although unsupervised learning methods do have important uses in connectionist systems, e.g., for dimensionality reduction of the input (Rumelhart and Zipser, 1985).

The first layer in Figure 1.3b is called the *hidden layer* and its units are called *hidden units* (Hinton and Sejnowski, 1983) to stress the fact that the critic is usually assumed to be able to evaluate the behavior of the output layer but generally knows nothing of the desired behavior of the first layer. For some tasks, multilayer systems can be defined for which the critic is able to instruct every layer during learning, but our interests are in learning methods for multilayer systems that can be applied to tasks for which such a critic does not exist.

A single-layer system can be severely limited in the type of mapping from input to output that it can implement. If linear threshold units are used, only those mappings that satisfy the linear-separability condition are possible. When the input representation and desired mapping do not satisfy the linear-separability condition, a hidden layer, or layers, can be added to learn new features that result in a new representation with which the necessary linear discriminations can be made.

The term "feature" is used to refer to the output function of a hidden unit. The outputs of hidden units can also be called the *terms* of the last layer's input representation since they become terms in the equation for the response of an output unit. However, "terms" is often used to refer to expressions in a symbol-based language, and "new terms" has acquired special significance through discussions of the *problem of new terms* in AI. While many similarities exist between the new term problem and the problem of learning new features (see Chapter II), "features" to some elicits a numerical perspective in accord with the numerical nature of computations in connectionist systems.

In at least one way, the use of "features" is misleading. Features are often considered as representing some part of a real or abstract entity that is intelligible to a human, i.e., its physical meaning and relationship to other features is easily grasped (e.g., a color or a shape). This is not always true of the output functions learned by a hidden unit (see, for example, the results of Chapter VII). It is more appropriate to consider hidden units to be learning intermediate-level concepts between the input-level features and the output-level decisions (using terminology from Fu and Buchanan [1985]), with no requirement that intermediate-level concepts be meaningful to humans. Sometimes, however, the features developed in a multilayer connectionist system will simplify the arduous task of deciphering what the system has learned.

Alleviating the linear-separability constraint by learning new features in hidden units is more desirable than using more-complex output functions in a single layer. This is true for two reasons. First, unless the desired form of the output function can be determined a priori, the more-complex output function might still be incapable of forming the needed discriminations and thus would require additional layers to learn new features. The second reason is that the relatively simple functions based on weighted sums of inputs lead to straightforward learning algorithms. As mentioned in the next section, some learning algorithms adjust weights according to the gradient of a performance criterion with respect to the weights, and the computation of such a gradient is much simpler for weighted-sum output functions than for other nonlinear functions.

The particular structure of the network in Figure 1.3b is used throughout this thesis for the following reason. With the network input vector passed through to all layers, the output layer can quickly form simple functions of the system's input before any learning occurs in other layers. If these functions are not sufficient for generating the correct output, then more complex functions can be constructed by the hidden units learning new features that correct the system's output. This structure might balance the tradeoff between the need for complex input-output mappings and the need for rapid learning, though this has not been analyzed.

Another desirable aspect of multilayer structures is the transfer of learning that results from hidden units learning new features. Transfer in connectionist systems occurs between similar inputs, where similarity depends on the input representation and on the manner in which inputs interact in the units' output functions. New features change input similarities by changing the representation of the input. The consideration of transfer resulting from new features is analogous

to Fu and Buchanan's (1985) attention to the generalization resulting from the learning of concepts at intermediate levels.

1.2 Credit Assignment in Multilayer Systems

A credit-assignment problem arises when a sequence of decisions are made before an evaluation is received. The problem is one of apportioning the final evaluation among the preceding decisions (Minsky, 1963). This statement encompasses two aspects of the credit-assignment problem, called the *temporal* and *structural* credit-assignment problems by Sutton (1984; Waterman [1970] also distinguished these classes). The temporal problem arises when a learning system produces a sequence of actions before an evaluation becomes available, such as the sequence of moves from the start of a chess game to the final win, draw, or loss, i.e., it is very difficult to correctly attribute credit or blame to the individual moves. Since many connectionist learning methods assume the existence of a teacher that provides performance feedback for every output of the system, the temporal credit-assignment problem is usually not addressed. Hampson (1983) and Sutton (1984) present approaches to the temporal credit-assignment problem that fit into the connectionist framework.

Once credit has been appropriately assigned to each decision in the sequence, the structural credit-assignment problem must be addressed. Many parts of a learning system might play a role in the decision that a particular move should be made in a chess game. The credit for that particular move must be apportioned among those parts. Credit or blame must be distributed throughout the *structure* of the learning system, in contrast to its distribution throughout the *temporal* sequence of actions.

The difficulty of the structural credit-assignment problem has retarded the development of multilayer learning systems. A change in one layer can enhance, reduce, cancel, or even reverse the effects of a change in another layer. The difficulty of this problem is attested to by the scarcity of multilayer learning systems reported in the literature. Smith, Mitchell, Chestek, and Buchanan (1977) review a number of learning systems, and only cite Samuel's (1959) and Uhr and Vossler's (1961) as examples of multilayer learning systems. These and other more recent multilayer learning systems are reviewed in Chapter III.

Despite the difficulties, research on the structural credit assignment problem has continued, and a number of approaches (depending on where distinctions are drawn) have been proposed. In the review of Chapter III, we classify methods according to the following general approaches:

- gradient methods, exact and approximate;
- methods based on a minimal-change principle;
- methods based on the measurement of the worth of the function computed by a unit.

If the output functions of the units of a connectionist system are differentiable with respect to the weights and a differentiable criterion can be defined as a function of the desired output and actual output, then the derivative of the criterion function with respect to each weight can be used to determine new weight values. This is the exact gradient method since the actual gradient (or, often, a sample of it) is used to update the weights.[3] Without differentiable output functions, approximations to the gradient can be made. The minimal-change principle is applied by methods that seek weight changes resulting in the smallest total change in weight magnitudes that remove output errors. Another method for assigning credit is by a measure of worth, which is often a function of the weights connecting hidden units to output units.

In the various methods, what constitutes credit varies. The terms *credit* and *blame* denote a scalar measure of responsibility, but some methods for structural credit assignment, such as the gradient methods, result in the assignment of errors to every weight. Methods that assign a

[3]Technically, an exact gradient method would not update the weights until all inputs have been presented and the sample gradients corresponding to each input have been summed to obtain the true gradient. Here we refer to methods that update weights after computing each sample gradient as exact gradient methods to distinguish them from methods for which even a sample gradient cannot be computed but must be estimated.

measure of worth to an entire unit are often used to generate new units, i.e., new weight vectors, rather than to assign errors to weights. The method by which new units are generated can either be *data-directed*, where the new weights are based on recent inputs, or *model-directed*, where the weights of current units of high worth are used as hypotheses as to which weight values are useful.

Learning systems based on symbolic forms of knowledge representation use some of these methods, such as minimal-change and worth, but often use other approaches to structural credit-assignment that rely on explicit domain knowledge. For example, one way in which Waterman's (1970) poker-betting production system assigns credit is by means of a prespecified decision matrix relating betting decisions and game-state variables.

In Chapter IV, some of the algorithms for the structural assignment of credit are compared with respect to their performance on a single task, reported in Chapter IV. Included in the comparative study is a novel algorithm that combines aspects of data-directed generation with weight adjustments based on approximate gradients. The algorithm is motivated by the fact that for gradient-based methods, little learning occurs in a hidden unit until the unit acquires an influence on an output unit, indicated by an interconnection weight of a sufficiently large magnitude. When the unit has little influence, the algorithm uses information about the system's performance to generate large jumps in weight values in the direction of input vectors for which the system's performance is poor. Thus a data-directed search is conducted for weights that define new features that characterize input vectors to which the system responds incorrectly. It is assumed that poor performance is a result of an insufficiently refined representation. Once a unit has developed an influence on an output unit, it uses an approximate gradient technique to refine the weights through minor adjustments.

1.3 Reinforcement Learning

One form of credit is a scalar evaluation, where more desirable actions yield higher evaluations than less desirable actions. To maximize the probability of receiving high evaluations, a learning system must generate alternative outputs and alter output probabilities based on the resulting evaluations so as to increase the probability of receiving high evaluations. This type of search has been called *reinforcement learning*, where the scalar evaluation is referred to as a reinforcement. Reinforcement-learning techniques have been developed in a number of disciplines, such as mathematical learning theory (Bush and Estes, 1959), learning automata theory (Narendra and Thathachar, 1974), reinforcement-learning control (Fu, 1970; Mendel and McLaren, 1970), and in early work with reinforcement learning in multilayer networks (Farley and Clark, 1954; Minsky, 1954). Barto and colleagues combined ideas originating in work with models of neurons as self-interested agents (Klopf, 1972, 1982), associative memories (Barto, Sutton, and Brouwer, 1981), animal learning theory (Sutton and Barto, 1981), learning automata (Barto and Anandan, 1985), and reinforcement-learning control (Barto, Sutton, and Anderson, 1985) to develop several classes of reinforcement-learning algorithms.

Some approaches to reinforcement learning deal with probabilistic dependencies between a system's output and the reinforcement it receives, such as the dependencies present in a connectionist system in which every unit receives the same reinforcement signal. The probabilities of reinforcement values that are correlated with a unit's output values would be much different for units directly responsible for the reinforcement, whereas a hidden unit would observe little difference before it has acquired significant connection weights to output units. Initial experiments with reinforcement learning algorithms in multilayer networks have been promising (Anderson, 1982; Barto, 1985; Barto and Anderson, 1985; Barto, Anderson, and Sutton, 1982). The review of Chapter III includes reinforcement learning methods, and in Chapter IV several reinforcement learning algorithms are included in the comparative study of learning algorithms for hidden units.

A difficult problem in reinforcement learning is the temporal credit-assignment problem described above. Sutton (1984) defined and analyzed a method for dealing with delayed reinforcement that he called the Adaptive Heuristic Critic (AHC) algorithm. This learning algorithm results in a linear function that predicts for a given state of the learning system's environment (given as input to the system) a sum of future reinforcements. The AHC algorithm is a generalization of Samuel's (1959) method for learning evaluation functions.

The utility of the AHC algorithm was demonstrated by using its output as the reinforcement

for a reinforcement-learning unit that learned associations from the states of a dynamical system to control actions (Barto, Sutton, and Anderson, 1983). The dynamical system involved a pole hinged to a cart, and the learning system's control actions were forces upon the cart. The learning system consistently learned to maintain the pole's balance. The goal of this task was expressed solely through a reinforcement value of -1 whenever the pole fell past a designated angle from vertical or the cart hit the end of its track, with zero-valued reinforcement all other times. (See Chapter VI for further details.)

In Barto, et al.'s, paper, the AHC and the reinforcement-learning algorithm for the pole-balancing task were each formulated as single connectionist units, using output functions based on a weighted sum of their inputs. To overcome the restrictions caused by the linearity of the units, the cart-pole system's 4-dimensional state space was "decoded" into 162 discrete regions (nonoverlapping 4-dimensional rectangles), and each state was represented by a 162-component standard unit basis vector (all zeros with a one in the position corresponding to the region of the state space in which the current state appears). Ballard (1984) calls this a *value-unit* encoding—each component of the input represents a particular range of values for, in this case, each dimension of the state. The result is a form of table look-up scheme with an entry in the table for every region of the state space.

The need for the special representation afforded by the decoder is removed by adding hidden units and a hidden-unit learning algorithm to both the reinforcement-prediction and force-generation parts of the controller. This is the subject of Chapter VI, in which results are presented from experiments with one and two-layer networks directly receiving the real-valued cart-pole state variables as input. It is shown how the development of new features by the two-layer networks results in a solution to the balancing task, whereas the one-layer networks do not find a solution. This connectionist system qualifies as a method for learning strategies for problem solving, as described below.

1.4 Problem Solving

Cohen and Feigenbaum (1981) define problem solving as "the process of developing a *sequence* of actions to achieve a goal." Problem-solving tasks entail a *search* over possible states of the problem to determine which state transitions and actions most directly lead to satisfaction of the goal. Knowledge used to constrain this search is called *heuristic* knowledge, typically expressed as heuristic rules relating particular states and recommendations for or against actions.

For a problem-solving system to improve its performance with practice it must learn appropriate search heuristics, a process referred to as *strategy learning* or strategy acquisition (see Keller [1982] and Langley [1983] for surveys of approaches to strategy learning). Langley (1985) lists three essential ingredients for a learning system to improve its search strategies. A learning system must first *generate* behavior in order to observe relationships among states, actions, and degrees of success in achieving the goal. Some initial search strategy must be employed to explore the effects of various actions on different states. Degrees of success are indicated by the second ingredient, a method for the *assignment of credit* or blame, which evaluates either individual actions, or sequences of actions. Only temporal credit assignment is explicitly considered; structural credit assignment is part of the modification process. The third ingredient is the *modification* of the search strategy, guided by the assigned credit or blame.

Each ingredient is characterized by several approaches, as indicated by the following outline:

- action generation (heuristically-guided search strategies)
  - breadth-first search
  - depth-first search
  - best-first search
  - probabilistic search
- temporal assignment of credit

  - complete solution paths
  - heuristic rules
  - evaluation functions, fixed or adaptive
- modification
  - of action-selection process
  - of evaluation function

Langley (1985) presents an adaptive production system for learning search strategies that features breadth-first search with complete solution paths and heuristics for credit assignment. The system was demonstrated by applying it to six puzzles, including the Tower of Hanoi puzzle.

The pole-balancing problem described in the previous section is a problem-solving task since a sequence of actions is desired that avoids certain regions of the state space. Credit assignment is performed by the adaptive evaluation function. The generation of actions is a probabilistic search that evolves into a best-first search as the evaluation function develops. Modifications are made to both the network that learns the evaluation function and to the force-generation network. Thus, the connectionist solution to the pole-balancing task (Barto, et al., 1983) is an approach to the learning of strategies, albeit with simpler representations than used in typical strategy learning systems that represent knowledge symbolically.

To compare and contrast this connectionist approach with Langley's approach, the connectionist system is applied to the Tower of Hanoi task, as described in Chapter VII. The representation issue of learning new features is investigated by using a representation of the puzzle's state that requires the system to learn new features in order to learn the puzzle's solution. Results show that the solution of the task is successfully learned by a two-layer network but not by a one-layer network. It is important to note that the goal of these experiments was not necessarily to demonstrate better performance than other strategy learning methods, but to study the combination of hidden-unit learning algorithms with reinforcement-learning algorithms to learn problem-solving strategies.

1.5 Research Objective and Method

The objective of the research reported in this dissertation is to bring together two of the most active research frontiers in connectionist learning—learning by hidden units and reinforcement learning. The first frontier faces the structural credit-assignment problem of learning in multilayer networks. Diverse approaches to this problem have been proposed over the years and have been presented using terminologies having little in common with one another. Much excitement has resulted from the success of recently-devised learning algorithms, but little effort is being expended in relating recent and past methods. Therefore, a secondary objective of this research is to compile a review of multilayer learning algorithms, presented in Chapter III. A framework of methods for structural credit-assignment is used to relate the learning algorithms by their approach to the structural credit-assignment problem. In addition to this review, recent algorithms as well as some from the past are compared in performance on an abstract task.

The second frontier is the development of learning algorithms for cases in which desired outputs are not known, as in problem-solving tasks. Reinforcement-learning techniques for connectionist systems have been developed for such tasks. At the outset of this work, it was unclear how compatible the reinforcement-learning algorithms and algorithms for learning in multilayer networks would be. Our goal was to shed some light on this issue by investigating one possible synthesis of these algorithms and demonstrating the potential of the approach by applying it to two strategy-learning tasks that, at least on the surface, seem to have considerably different natures.

In fulfilling these objectives, the following steps were taken:

1. issues were studied concerning the learning of new features, and comparisons were drawn between the learning of features in connectionist representations and the learning of new terms in symbolic representations;

2. learning algorithms for multilayer systems were reviewed;

3. learning algorithms were compared in performance on a task formulated in such a manner as to require new features;

4. a hidden-unit learning algorithm and a reinforcement learning algorithm were synthesized to form a connectionist approach to strategy learning.

The comparison of learning algorithms is a subtle endeavor. The algorithms tested are governed by from one to six parameters, and fair comparisons must involve the optimization of these parameters for each algorithm. In optimizing the parameters, the experimenter effectively becomes part of the learning loop, and the ease or difficulty with which good parameter values are found partially determines the robustness of the algorithm. Aspects of the task must also be varied, such as the sequence of inputs, to avoid biasing the results in favor of algorithms that only work well for certain versions of the task. Another confounding factor is that some of the learning algorithms are stochastic in nature, and must be compared with deterministic algorithms. Finally, algorithms must be compared using more than one measure of performance; the final performance level at the conclusion of an experiment does not reflect the cumulative performance and thus the speed with which good performance is achieved. These issues add up to a need for a large set of repeated experiments to obtain statistically significant results and fair comparisons. The computation time required for such a schedule necessitated the limited number of tasks used in this dissertation—a single task for the comparison of learning algorithms and two tasks for the demonstration of the learning of strategies.

Conclusions are thus restricted to the particular tasks used in the experiments, but the originality of the approaches taken for the comparative study and for the learning of strategies by connectionist systems makes these initial steps of interest to all who are curious about the capabilities of connectionist systems. To extend the conclusions of this work, very important questions regarding scaling problems must be addressed. Studies are needed on how the performance of connectionist learning algorithms changes as the number of inputs and outputs and the number of units increase and as tasks get more difficult.

1.6 Reader's Guide

The chapters of this dissertation are divided roughly into those dealing with learning in multilayer networks and those describing connectionist systems for strategy learning. Apart from these chapters, Chapter II contains a discussion of the ways in which the learning of new features can facilitate learning in connectionist systems. Simulations of learning in single-layer networks illustrate potential advantages of learning new features. Details of the results are in Appendix A. In addition, relationships between new features in connectionist representations and new terms in symbolic representations are discussed.

The review of methods for learning in multilayer systems is presented in Chapter III. Attention is limited to systems based on numerical representations. The last section of Chapter III contains an outline (modeled after that used by Smith, Mitchell, Chestek, and Buchanan [1977]) of the systems reviewed in the chapter. This can be skimmed independently of the rest of the chapter for an understanding of how the learning systems are characterized in the review. Chapter IV contains the results of the comparative study, including graphs of performance in the form of superimposed learning curves for easy comparison. On first reading it is expedient to skim the specifications of the algorithms.

Algorithms for the application of multilayer connectionist systems to the learning of strategies are developed in Chapter V. The two tasks used to demonstrate strategy learning with connectionist systems are the pole-balancing task and the Tower of Hanoi puzzle. Experiments with these tasks are presented in Chapters VI and VII, respectively. Results are discussed in light of the performance of symbolic strategy-learning methods on similar tasks.

Finally, in Chapter VIII, general conclusions are drawn and some implications of this research are discussed. Directions for future research are also indicated.

Chapter 2

New Features and the Facilitation of Learning

A representation consists of a set of concepts and the means by which they are used to make decisions. A *concept* is defined as a partition of the space of possible inputs to a learning system (after Utgoff [1986]), whether the space is composed of discrete entities labeled with symbols meaningful to a human, or is composed of continuous dimensions encoding numerical features. The goal of inductive learning (a term encompassing the kinds of learning studied in this thesis) is to search the space of possible concepts for a set of concepts that best satisfies some criterion (Mitchell, 1982). A system that learns concepts from examples must find the concept among all concepts expressible in the representation language that are consistent with the largest proportion of training examples. A system that learns by discovery (guided, for example, by an evaluation function) must find the set of concepts that results in the largest expected evaluation. For such a learning process to be effective for large input spaces, *bias* must be introduced to constrain the search (Utgoff, 1986). Utgoff outlines several kinds of bias and studies one—restrictions on the space of possible concepts—in some detail. In this chapter we examine such restrictions in connectionist representations and how they are dealt with by learning in hidden units. Relationships to similar issues arising in learning with symbolic representations are discussed.

In the following discussion, terms, features, and concepts are considered to be equivalent on an abstract level—instances of each specify partitions of a system's input space. For the most part, we follow tradition by referring to concepts represented symbolically as *terms* and concepts represented numerically, as in connectionist systems, as *features*.

The effects of new features on learning are illustrated by means of several simulations involving three learning tasks and single-layer connectionist learning systems. Learning performance is observed for several different input representations to show the effects of inappropriate bias and how it is shifted by adding new features that:

1. are required for the task's solution, i.e., are *missing* from the initial input representation, or
2. are not required but that result in *beneficial* generalization, or transfer of learning among input vectors.

The first illustration pertains to Case 1 and the other two to Case 2. The results of these simulations demonstrate the potential gains in performance of a system capable of learning new features. The simulations are presented in detail since much of the discussion is relevant to later chapters.

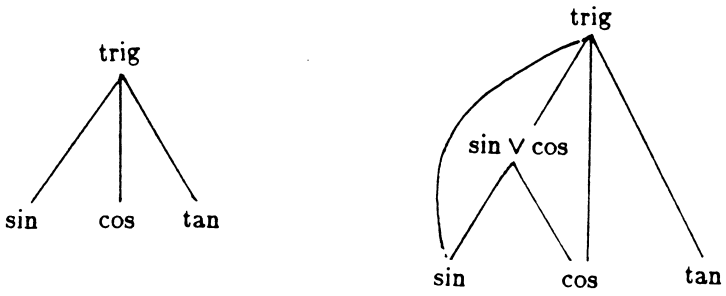


Figure 2.1: Adding a New Term to a Symbolic Representation

2.1 The Problem of Missing Features

Before presenting the simulations of connectionist learning systems, similarities between the issues of new terms and new features are illustrated by describing the new term problem studied by Utgoff (1986) in his work with the LEX system. Then an artificial situation is described in which a connectionist system faces the same problem. The point of this example is the parallel nature of the missing term and missing feature problem. It is important to note that not all symbolic representations can be directly related to a connectionist form. The variants of the missing term and the missing feature problems are, however, sufficiently analogous that learning methods arising from both camps must deal with the same issues.

2.1.1 Missing Terms in LEX

The LEX system of Mitchell, Utgoff, and Banerji (1983) learns to perform symbolic integration through the development of concepts that classify expressions according to whether or not particular operations have been found useful in simplifying a particular integral expression. A *generalization hierarchy*, implicitly defined by a grammar that specifies how terms can be combined, constrains the search for concepts by restricting generalizations and specializations to only those resulting in concepts included in the hierarchy.

A small portion of the generalization hierarchy for trigonometric functions is shown in Figure 2.1a. During the course of learning, LEX discovered that the "integration-by-parts" operator should be applied to certain expressions containing either the term *sin* or the term *cos*, but not to expressions containing the term *tan*. However, the only available generalization of the *sin* and *cos* terms is *trig*, which does not exclude the *tan* term. To make the required generalization, the new term $sin \vee cos$ must be added and the generalization hierarchy altered as in Figure 2.1b. Utgoff (1986) proposed the "least disjunction" method for automating this process.

This example is continued by recasting the problem as a difficulty faced by a connectionist system. Let us assume that the system contains a linear threshold unit whose output is used by subsequent units to decide when to apply the "integration-by-parts" operator, and this unit must indicate either the presence (an above-zero output) or the absence (a below-zero output) of the *sin* term or the *cos* term in an expression. Let us also assume that the trigonometric terms are represented by two-component vectors as follows:

x_1, x_2	represents
(1, 1)	*sin*
(2, 1)	*tan*
(3, 1)	*cos*
(∗, 1)	*trig*

The symbol "∗" is a "don't care" symbol. A linear threshold unit is only capable of linearly splitting the two-dimensional space containing these vectors—the region in which the unit's output

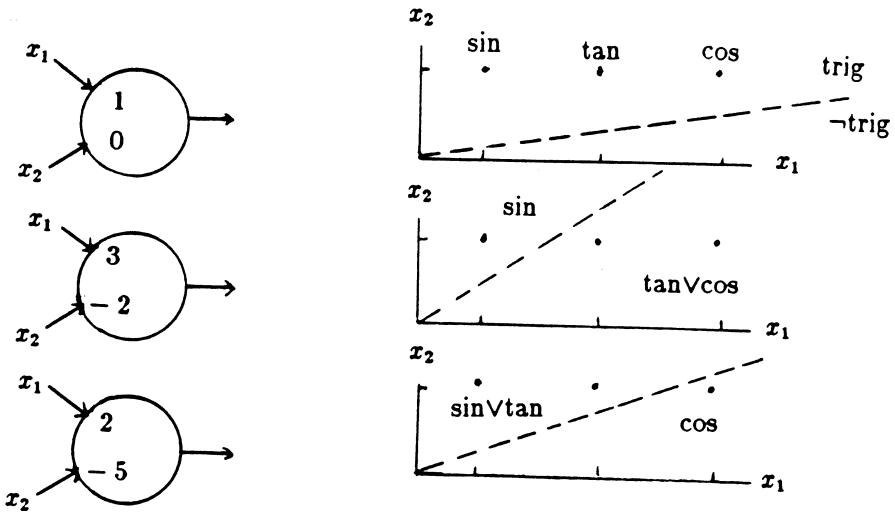


Figure 2.2: Possible Concepts using Connectionist Representation

is above zero is separated by a line from the below-zero region. As shown in Figure 2.2, with this representation the concepts *trig*, $\neg trig$, *sin*, *cos*, $tan \vee cos$, and $sin \vee tan$ can be represented by a single unit, but $sin \vee cos$ cannot. (Examples of weight values that result in these linear separations are included in the figure.) Other value assignments for x_1 and x_2 exist for which $sin \vee cos$ can be represented, but for any two-dimensional vector representation some concepts cannot be expressed.

To solve this problem the representation must be augmented with an additional component. This is achieved by adding a new unit, such as the unit in Figure 2.2c (or by adjusting the weights of a unit already present to be similar to those of the unit in Figure 2.2c). The new unit's output becomes a new input component to the existing unit. This structure, shown in Figure 2.3, results in the new representation:

x_1, x_2, x_3	represents
(1, 1, 0)	*sin*
(2, 1, 0)	*tan*
(3, 1, 1)	*cos*
$(*, 1, *)$	*trig*

A linear threshold unit separates this three-dimensional space with a plane, and due to the new feature, x_3, this plane can be placed such that *sin* and *cos* are in one region whereas *tan* is in the other. Thus, the two-unit system with the weight values shown in Figure 2.3 represents the concept $sin \vee cos$.

In the remainder of this section a set of experiments is presented in which a single linear threshold unit is applied to a learning task whose first formulation involves an input representation with which the single unit cannot solve the task. Different input representations are tried, including the addition of new features as functions of the original input components. A perceptron learning algorithm (Rosenblatt, 1962) is used in the single unit to update its weights. Other learning algorithms show similar differences in performance for the different input representations.

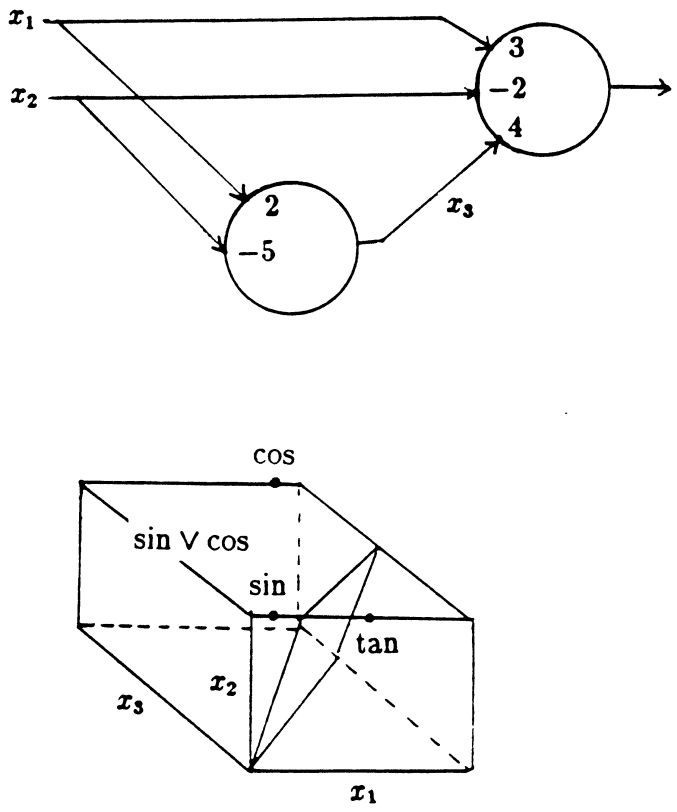


Figure 2.3: Adding a New Feature to the Connectionist Representation

2.1.2 The Multiplexer Task

One of the simplest examples of a problem having missing features is the two-input "exclusive-or" task presented to a single linear threshold unit. The input vectors, $x^{(i)}$, and corresponding desired outputs, $d_{x^{(i)}}$, are

$$x^{(1)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \; x^{(2)} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \; x^{(3)} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \; x^{(4)} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$d_{x^{(1)}} = \; 0 \;\; , \; d_{x^{(2)}} = \; 1 \;\; , \; d_{x^{(3)}} = \; 1 \;\; , \; d_{x^{(4)}} = \; 0.$$

The vectors $x^{(2)}$ and $x^{(3)}$ cannot be separated from $x^{(1)}$ and $x^{(4)}$ by a single linear discriminant function, and thus the exclusive-or function cannot be implemented by a single linear threshold unit. Higher-dimensional exclusive-or functions are referred to as "parity" functions and are one type of problem that Minsky and Papert (1969) used to discuss the limitations of the perceptron.

The task used for the following missing-feature illustrations is also used for the comparative study of Chapter IV. The two-input exclusive-or task was not used for the following reason. A system consisting of two layers with a single unit in each layer can implement the exclusive-or function. A very simple random search applied to the weights of the single hidden unit performs very well because the probability of randomly choosing a solution weight vector is relatively large.[1] A more difficult task was desired to tax the random search and other methods that directly search a network's weight space.

A *multiplexer function* of four data bits and two address bits was chosen as the function to be learned. The input vectors consist of two address bits and four data bits, plus a constant component of value 0.5. To solve this task, a learning system must route to the system's output the binary value of the data bit that is addressed by the two address bits. Let us call the address bits a_1 and a_2 and the data bits d_1, d_2, d_3, and d_4. A minimal logical expression for the multiplexer function is

$$\bar{a}_1 \bar{a}_2 d_1 \vee \bar{a}_1 a_2 d_2 \vee a_1 \bar{a}_2 d_3 \vee a_1 a_2 \; d_4.$$

To prove that this task is not linearly-separable, and thus not solvable by a single-layer system of linear threshold units, it suffices to show a subset of the input vectors whose members cannot be associated with the desired outputs. Consider the input vectors and desired outputs below:

input vector	desired output
(0,0,0,1,0,0,0.5)	0
(0,0,1,0,0,0,0.5)	1
(0,1,0,1,0,0,0.5)	1
(0,1,1,0,0,0,0.5)	0

The associations between By ignoring the constant input components and studying any two remaining components, we see that the relationship between the components and the desired output is an exclusive-or function. Thus, the multiplexer function cannot be implemented by a single linear threshold unit. Ways in which a multilayer system might solve this task are discussed below through examples of different, fixed representations of the input vectors.

One more reason for selecting the multiplexer function was our desire for a task that requires more than one new feature, i.e., more than one hidden unit. Algorithms for which hidden units learn similar features might perform satisfactorily when only one new feature is required, but would fail when two or more are needed. The multiplexer task is sufficiently complex that more than one new feature is needed. The results of our experiments with multilayer learning systems and the multiplexer task suggest that a minimum of three hidden units are required, although this has not been proved.

In the next two sections, the perceptron learning algorithm and two performance measures are defined, after which the experiments and results are discussed.

[1] The probability of finding a solution is equal to the ratio of the combined volumes of the regions containing solution weight vectors to the total volume of the weight space.

2.1.3 The Perceptron Learning Algorithm

Let X be the set of all input vectors and x be a single input vector from that set. The output of a linear threshold unit, for a given input vector x and weight vector w, is

$$y_x = \begin{cases} 1, & \text{if } \sum_i w_i\, x_i > 0; \\ 0, & \text{otherwise.} \end{cases}$$

Let the desired output for x be d_x and let X_w be the subset of X containing all input vectors for which the output, given w, is not the desired output, i.e.,

$$X_w = \{x \,|\, y_x \neq d_x\}.$$

The perceptron learning algorithm is a gradient-descent procedure that minimizes the perceptron criterion function, $J(w)$, where

$$J(w) = \sum_{x \in X_w} |d_x - y_x|.$$

During training, the values of x, d, y, and w for each time step generate the sequences $x[t], d[t], y[t]$, and $w[t]$. Each time step consists of:

1. Specification of input $x[t]$ and desired output $d[t]$,
2. Calculation of $y[t]$, using $x[t]$ and $w[t]$,
3. Calculation of new weight values, $w[t+1]$.

The new value for the i^{th} weight is calculated according to the following rule:

$$w_i[t+1] = w_i[t] + \rho\,(d[t] - y[t])\,x_i[t],$$

where ρ is a positive real number that controls the magnitude of the weight change. This is Rosenblatt's "α-perceptron, error-corrective, quantized" learning algorithm (Rosenblatt, 1962), and is an example of a fixed-increment error-correction procedure (Duda and Hart, 1973). The convergence of this algorithm to a solution, when a solution exists, has been proven in several ways (Minsky and Papert, 1969; Nilsson, 1965; Rosenblatt, 1962). The only conditions on the proofs are that $\rho > 0$ and that every input vector $x \in X$ occurs infinitely often.

2.1.4 Performance Measures

Repeated runs of a fixed number of steps were performed in the following experiments. Let the number of runs be r, with each run starting with zero-valued weights and trained for s time steps. Learning performance is measured by two quantities, μ and ν. The average number of cumulative errors per run, μ, is defined as follows. Let the number of errors (number of output units with incorrect output) for the t^{th} time step of run k be $e_k[t]$, defined as:

$$e_k[t] = \sum_{j \in O} |d_j[t] - y_j[t]|, \qquad \text{where } O = \left\{ \begin{array}{c} \text{indices of} \\ \text{output units} \end{array} \right\},$$

and where $y_j[t]$ is the output of the j^{th} unit at time step t and run k and $d_j[t]$ is that unit's desired output corresponding to input $x[t]$. Summing the number of errors over the s time steps and r runs and multiplying by a scale factor of $1/r$ results in the following equation for μ:

$$\mu = \frac{1}{r} \sum_{k=1}^{r} \sum_{t=1}^{s} e_k[t].$$

For a system of m output units, the value of μ ranges from 0 to $s \cdot m$, the maximum number of errors in a single run. If no errors are made $\mu = 0$, signifying perfect performance,[2] and $\mu = s \cdot m$ means every output was in error.

Note that μ is a cumulative measure of performance that depends on the errors at every time step. A dramatic increase in performance toward the end of a run is not reflected in the value of μ. A second performance measure is needed that depends only on the final performance level achieved at the end of a run. This measure, called ν, is calculated as follows. The final weight values at the end of a run are frozen while the entire set of n ($n = 64$ for the multiplexer task) input vectors are presented one at a time. The system's output for each input vector is compared to the desired output and the errors tallied. For run k, the expected value, h_k, over all input vectors of the number of errors is given by:

$$h_k = \frac{1}{n} \sum_{x \in X} \sum_{j \in O} |d_{xj} - y_{xj}|,$$

where d_{xj} is the desired value of output component j for input x, and y_{xj} is the actual output given input x when the weight values at the end of run k are used to compute the output. Since the multiplexer experiments involve a single output unit ($||O||$=1), this expression could be simplified. The possibility of more than one output unit is included to make this definition applicable to other experiments in this chapter. Summing the number of errors for r runs and dividing by the number of runs results in

$$\nu = \frac{1}{r} \sum_{k=1}^{r} h_k,$$

which ranges from 0 to $m \cdot n$, the number of output units times the number of input vectors. Since ν depends only on the performance achieved at the end of each run, it indicates the reliability with which a solution to the task was found. On the other hand, μ is a measure of the speed with which the solution was found.

2.1.5 Original Representation

The four data bits, d_1, d_2, d_3, d_4, and the two address bits, a_1, a_2, and a constant input of 0.5 constitute the unit's input vector. The single unit used in the experiments with the original representation of the multiplexer task is shown in Figure 2.4.

2.1.6 An Ideal Representation

Since the classes of input vectors requiring the same output are not linearly-separable, a solution does not exist for the single linear threshold unit. Now we can ask how the representation of the input vectors might be changed to improve learning performance on this particular task. A number of representations can be used, differing in the amount of information assumed known about the task before learning takes place. If complete knowledge of the task is assumed, then the representation

$$x^{(i)\prime} = d_{x^{(i)}}$$

would be ideal. (We use $x^{(i)\prime}$ rather than $x^{(i)}$ to indicate the input vectors of a representation other than the original representation.) In this case, the value of each $x^{(i)\prime}$ is actually the value of the corresponding desired output $d_{x^{(i)}}$. The unit must only learn to generate the value that is provided as input, obviously an easy task.

Our interests, however, are with the performance of learning algorithms on tasks for which the amount of a priori knowledge is a great deal less than that required for defining such ideal representations. Even when parts of a task are sufficiently well-understood, tailoring an otherwise fixed representation for those parts could be disastrous. For example, consider a problem for which

[2]Perfect performance can only be achieved if the weights at the start of every run happen to be correct.

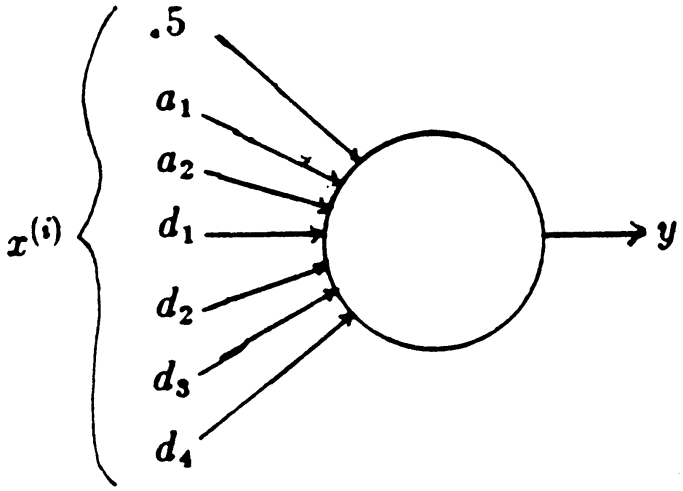


Figure 2.4: Multiplexer Task—Original Representation

each output of the system consists of two binary components. Assume there are five input vectors and their desired outputs are

$$d_{x^{(1)}} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, d_{x^{(2)}} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, d_{x^{(3)}} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, d_{x^{(4)}} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \text{ and } d_{x^{(5)}} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Let us say that the designer of a learning system (composed of a single linear threshold unit) is aware of the value of the first component of the desired output for all input vectors, but knows nothing of the second. The ideal representation, given this knowledge, is

$$x^{(1)} = (0),\ x^{(2)} = (1),\ x^{(3)} = (0),\ x^{(4)} = (1), \text{ and } x^{(5)} = (0),$$

where the input is exactly the first component of the desired output. However, with this representation, the task of learning the second component is impossible. The input classes defined by the second desired output component are $\{x^{(1)}\}$ and $\{x^{(2)}, x^{(3)}, x^{(4)}, x^{(5)}\}$ which cannot be linearly-separated, i.e., a linear threshold unit cannot implement the mapping from each x_i to the second output component.

2.1.7 A Representation Resulting in No Generalization

In the above example, the transfer of learning from one input vector to another is extremely helpful for learning the first output component, but it prevents learning of the second component. A representation that avoids this problem arises from the extreme strategy of preventing all generalization, i.e., no transfer between input vectors can occur. For linear units, this implies that the input vectors be standard basis vectors. Using standard unit basis vectors, the input representation for the multiplexer task becomes:

$$\begin{aligned} x^{(1)\prime} &= (0,0,0,\ldots,0,0,1)^T, \\ x^{(2)\prime} &= (0,0,0,\ldots,0,1,0)^T, \\ x^{(3)\prime} &= (0,0,0,\ldots,1,0,0)^T, \\ &\vdots \\ x^{(63)\prime} &= (0,1,0,\ldots,0,0,0)^T, \\ x^{(64)\prime} &= (1,0,0,\ldots,0,0,0)^T. \end{aligned}$$

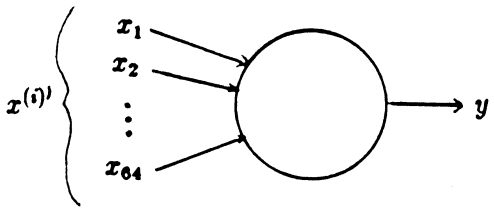


Figure 2.5: Multiplexer Task—Basis-Vector Representation

Each input vector has 64 binary-valued components, all being 0 except one, resulting in the structure shown in Figure 2.5. The input classes are certainly linearly-separable. In fact, any mapping from the 64 input vectors to the two possible output values, 0 and 1, is implementable by effectively filling in a table of 64 slots with the desired output values. But in removing all generalizations, helpful transfer disappears, thus slowing the speed of learning over that possible with the ideal representation and eliminating transfer to novel input vectors during performance.

2.1.8 New Features Added to Original Representation

The previous section showed a representation that guarantees the existence of a solution. In doing so, the possibility of generalizing from one situation to another was removed. We would like a new-feature algorithm—an algorithm for learning new features by modifying the weights in hidden units—to similarly guarantee that a representation will be formed with which a solution to the task exists, but to do so without removing beneficial generalizations. One approach to meeting these goals is to always include the components of the original representation along with any new features that are developed. New features add dimensions along which additional discriminations can be made, overcoming misleading generalizations.

Recall that a minimal expression for the solution to the multiplexer task is

$$\bar{a}_1\bar{a}_2d_1 \vee \bar{a}_1a_2d_2 \vee a_1\bar{a}_2d_3 \vee a_1a_2\ d_4.$$

The most straightforward way to add features that a) make the solution of this task possible, and b) can be formed by a single layer of hidden units, is to add features corresponding to the four disjuncts in the above expression, i.e.,

$$\begin{aligned} feature_1 &= \bar{a}_1\bar{a}_2d_1, \\ feature_2 &= \bar{a}_1a_2d_2, \\ feature_3 &= a_1\bar{a}_2d_3, \\ feature_4 &= a_1a_2d_4, \end{aligned}$$

resulting in the structure of Figure 2.6.

2.1.9 Results

Preliminary experiments with all three representations were run to determine the sensitivity of the perceptron learning algorithm to the parameter ρ. These results, summarized in Appendix A, show that the perceptron algorithm had little sensitivity to ρ. For each value of ρ, 30 runs of 50,000 time steps each were performed. Weights were set to zero at the start of every run. Input vectors were chosen randomly, without replacement, for each step: every input vector is presented once within the first 64 steps, a second time within the next 64 steps, and so on. This reduces the solution time from that needed for a completely random presentation of input vectors. However, we do randomize the order of presentation within each 64 step interval to remove any bias in the results due to a particular order of presentation.

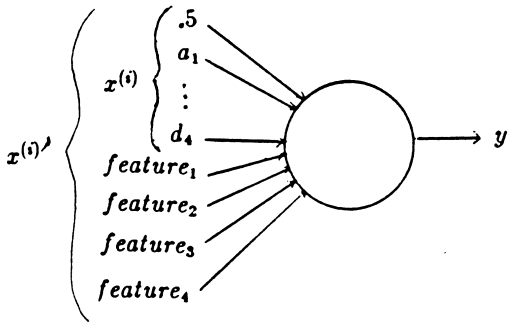


Figure 2.6: Multiplexer Task—Original Representation Plus New Features

Representation	ν	μ
Original	23.9 ± 0.88	208 ± 2.3
Basis Vectors	0.0 ± 0.0	32.0 ± 0.0
New Features	0.0 ± 0.0	31.7 ± 1.2

Table 2.1: Performance of Single Unit on Multiplexer Task

Further experiments were performed with $\rho = 1$. For each representation, 100 runs of 500 steps each were performed, giving the results shown in Table 2.1. The 99% confidence intervals are listed with each result to provide a quick check of statistical significance.[3]

With the original representation, the final performance level of the perceptron unit was $\nu = 23.9 \pm 0.88$, indicating that an average of approximately 24 of the 64 input vectors were incorrectly classified at the end of each run. When the basis vectors were used to represent the input, the task was solved for all runs, indicated by $\nu = 0.0 \pm 0.0$. With the initial weight values of zero, the output is correct for 32 of the 64 input vectors. No transfer occurs among input vectors represented as standard basis vectors, so at least 32 errors will occur during a run as the output is corrected for the 32 input vectors whose desired output is 1. No more than 32 corrections are needed, as shown by $\mu = 32 \pm 0$. The task was also reliably solved ($\nu = 0.0 \pm 0.0$) with the original representation plus the four additional features.

A better understanding of the relationships in learning performance for the different representations is gained by superimposing the resulting learning curves of errors versus time steps, as shown in Figure 2.7. The maximum number of errors per time step is one because there is only one output unit. To generate the curves, the number of errors per step was averaged over the 100 runs. Additional averaging of the values for 5-step intervals was done to further smooth the curves. With the original representation the error is never reduced to zero, since a single-unit solution to the multiplexer task does not exist. The learning curve does show that better-than-chance (0.5 errors per time step) performance was achieved.

The learning curve corresponding to the basis-vector representation shows a dramatic jump from 0.5 to 0 errors per time step. This is due to the fact that no generalization occurs between input vectors. As soon as every input vector that requires the output 1 is presented at least once, the error becomes zero.

The learning curve for the original representation plus the four new features initially decreases faster than does the curve for the basis-vector representation, showing that the generalization caused by the new features is beneficial in the early stages. However, a longer period of time is required to completely reduce the error to zero, indicating that some misleading generalization still occurs. The difference in the cumulative number of errors, μ, resulting from the use of the

[3]Given a sample's standard deviation, s, and size, n, its 99% confidence interval was calculated as $\pm 2.5758 s/\sqrt{n}$.

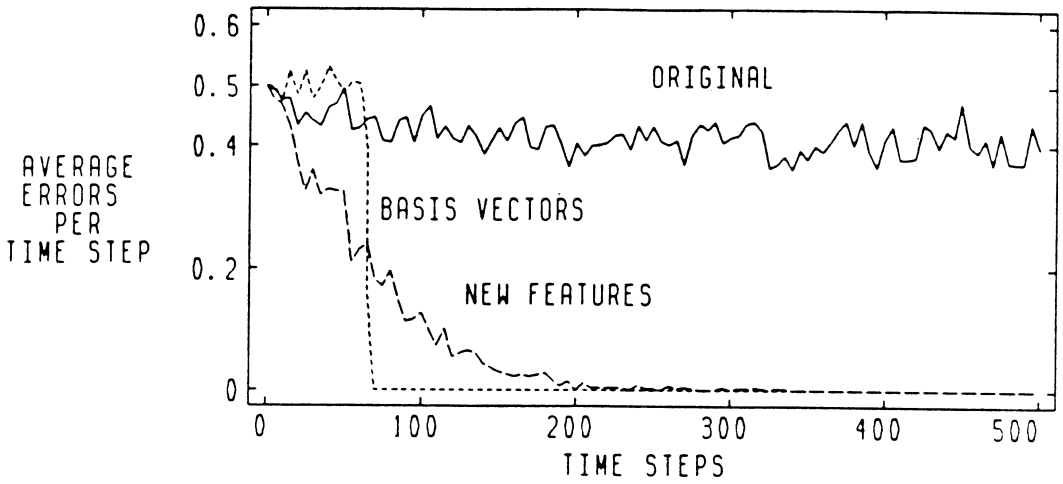


Figure 2.7: Multiplexer Task—Learning Curves

new-feature representation and the basis-vector representation is not significant (31.7 ± 1.2 versus 32.0 ± 0.0).

The following sections pertain to tasks that do not require new features but that can be solved quicker if appropriate new features are learned. Experiments with connectionist systems and two such tasks are presented, followed by a discussion of the similarities between the issues raised by these tasks and those described by Fu and Buchanan (1985) for learning in production systems.

2.2 New Features That Add Beneficial Generalization

A useful new-feature algorithm should be able to overcome misleading generalization caused by the absence of required features, but should also be able to add generalization where it might be helpful. In effect, it should increase the similarity of input vectors that require similar output, so that in the future the learning of new output values to these input vectors is facilitated.

This process can be viewed as a "lumping", or clustering, of the input vectors. The potential of this type of representation development has been well-recognized, and many measures of similarity have been proposed with which to determine cluster membership. Such measures are often based on a distance metric in the input space; input vectors separated by small distances are collectively represented by a cluster label, reducing the dimensionality of the input. Another measure, used by Fukushima (1973, 1980), is the closeness in time of the presentation of the input vectors. He developed a character-recognition system with some degree of rotational and translational invariance by presenting a character and its variants in sequence, excluding vectors derived from any other characters. In contrast, Rendell (1985) has experimented with a more goal-directed similarity measure. In learning an evaluation function, he clustered input vectors based on the similarity of their outcome in terms of success or failure in game-playing situations.

Rendell's use of the desired output to cluster input vectors can be adapted to learning by the hidden units of a connectionist network. The hidden-unit learning algorithm must have access to the system's desired output, which can be obtained directly from the environment with which the system is interacting or from the output layer of the system after it has solved the current task. To illustrate the possible results of following this strategy, experiments were run with a single layer of perceptron units and a task referred to below as the input-cluster task. Again, the input representation is varied to show the usefulness of new features.

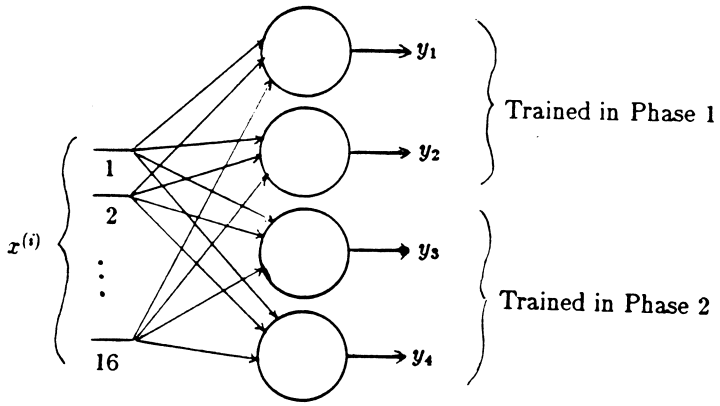


Figure 2.8: Input-Cluster Task—Original Representation

2.2.1 The Input-Cluster Task

To highlight the effects of adding beneficial generalization, we used standard unit basis vectors as the initial representation for this task so no generalization would be possible. The set of input vectors consists of 16 standard unit basis vectors and the system's output is a vector of four, binary-valued components, rather than the single component of the multiplexer task. The desired output vectors are defined as follows:

$$d_{x^{(1)}}, \ldots, d_{x^{(5)}} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}, d_{x^{(6)}}, \ldots, d_{x^{(11)}} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, d_{x^{(12)}}, \ldots, d_{x^{(16)}} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix},$$

i.e., the members of the input class $\{x^{(1)}, \ldots, x^{(5)}\}$ require output $(1, 0, 1, 0)^T$, the members of $\{x^{(6)}, \ldots, x^{(11)}\}$ require output $(1, 1, 1, 1)^T$, and the members of $\{x^{(12)}, \ldots, x^{(16)}\}$ require output $(0, 1, 0, 1)^T$.

The members of an input class initially have no similarity, i.e., no generalization takes place. A learning system would have an advantage if it developed features that indicate the class to which the current input vector belongs, thus "lumping" the members of each class. To make this advantage more obvious the task is divided into two phases. In Phase 1 only the first two output units are trained, and in Phase 2 only the last two are trained, the objective being to show how a process that forms new features during Phase 1 can facilitate learning during Phase 2.

2.2.2 Original Representation

For the first experiment the original representation was maintained throughout both phases. Any new-feature algorithm that was not successful in finding a better representation for this task should at least perform as well as the single-layer algorithms used in this experiment. A poorer performance level would mean that the new-feature algorithm was a deterrent to learning with the original representation. The structure of the network with the original representation is shown in Figure 2.8. The units trained during each phase are marked.

2.2.3 New Features—Basis Vectors as Class Labels

To correctly generalize among the input vectors, features must be added that are common to the members of an input class. There are many ways to do this, but we limited our attention to vectors of binary-valued components because we assume that the new features are to be generated

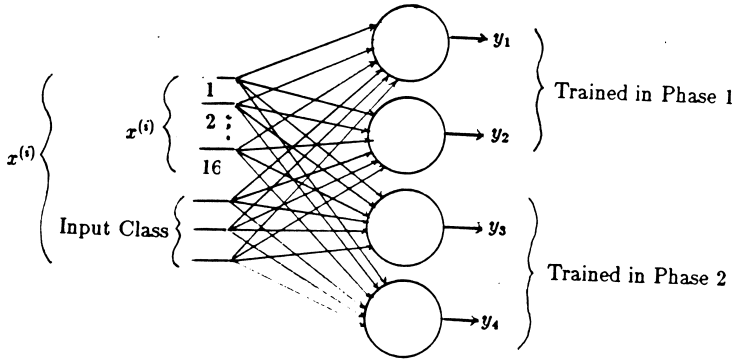


Figure 2.9: Input-Cluster Task—Basis Vectors as Class Labels

by hidden units whose outputs are binary-valued. One way to add features is simply to create a feature for each input class with one value for all input vectors in the corresponding class and the other value for all other input vectors. An example is a set of three-dimensional standard unit basis vectors, each representing a particular input class as follows: class $\{x^{(1)}, \ldots, x^{(5)}\}$ is represented as $(0, 0, 1)$, class $\{x^{(6)}, \ldots, x^{(11)}\}$ as $(0, 1, 0)$, and class $\{x^{(12)} \ldots, x^{(16)}\}$ as $(1, 0, 0)$. Adding these features to the original representation results in the structure shown in Figure 2.9.

The new features are not added until the completion of Phase 1 to simulate the development of features during Phase 1. The new features are present throughout Phase 2. The difference in the rates of learning for the two phases is due to the presence of the new features. It is possible that an algorithm for learning new features would do better than the performance level shown in this experiment by adding features early in Phase 1.

2.2.4 New Features—Class Labels with Generalization

Another possible encoding of the class labels is the set of two-dimensional vectors $\{(0, 1), (1, 1), (1, 0)\}$, each vector representing one of the three input classes. These labels are actually the desired output for the respective input classes. Now generalization will occur between input classes. The system's structure is shown in Figure 2.10.

2.2.5 Results

As was done for the multiplexer task, preliminary experiments were performed to compare performance for various values of ρ. These experiments consisted of 50 runs of 200 time steps each, 100 steps per phase. Again, varying ρ had little effect on performance, as seen by the results summarized in Appendix A. Using $\rho = 1$, an additional 50 runs were made for each representation, resulting in the performance values shown in Table 2.2. Every run solved this task within 100 steps, so the values of ν were all equal to 0 and do not appear in the tables of Appendix A. The cumulative error measure, μ, is decomposed into two values, μ_1 and μ_2, representing the number of errors tallied during the first and second phases, respectively. The maximum possible number of errors during a phase (the maximum value of μ) is

$$\begin{aligned} \text{number of output units being trained} \cdot \text{steps} &= 2 \cdot 100 \\ &= 200. \end{aligned}$$

On the average, a random output strategy would accumulate half the maximum number of errors, or 100 errors, giving values of $\mu_1 = \mu_2 = 100$.

The first experiments involved the use of the original representation for both phases. Since the representation of Phase 2 was identical to that of Phase 1, there was no significant difference

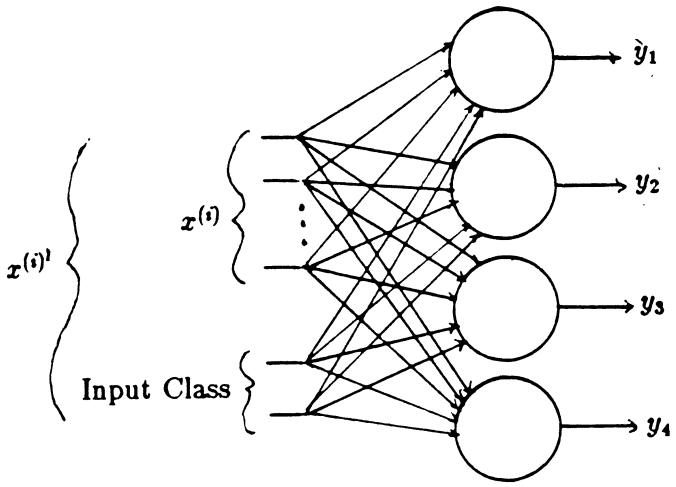


Figure 2.10: Input-Cluster Task—Class Labels with Common Feature

Representation	μ_1	μ_2
Original	22.0 ± 0.05	22.0 ± 0.00
Unique New Features	22.0 ± 0.05	4.0 ± 0.00
One Common New Feature	22.0 ± 0.05	3.0 ± 0.25

Table 2.2: Performance of Single-Layer System on Input-Cluster Task

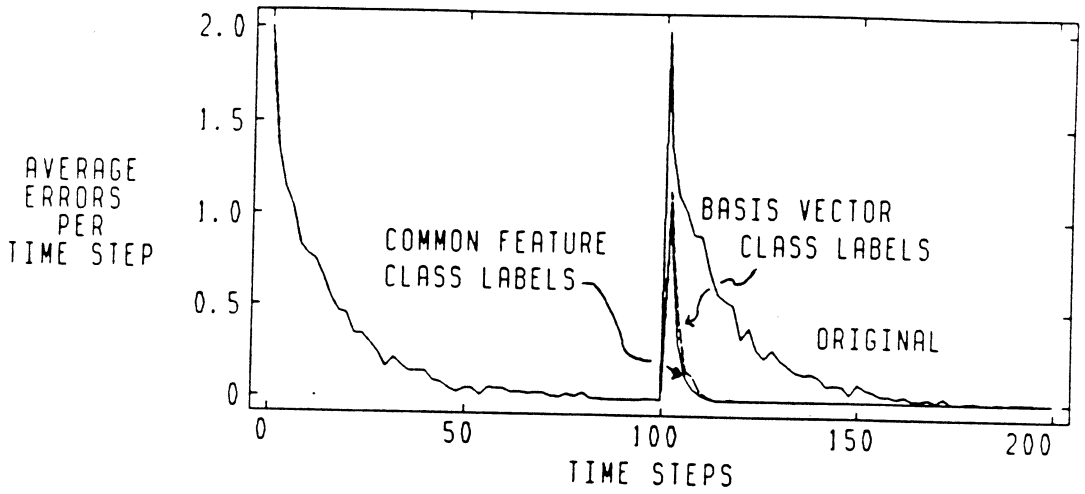


Figure 2.11: Input-Cluster Task—Learning Curves

in performance during the two phases. Adding new features at the start of Phase 2 that define basis vectors for each input class results in the increase in performance, or decrease in the number of errors, from Phase 1 to Phase 2 that we can expect from a successful new-feature algorithm. The number of errors incurred during Phase 2 was $5\frac{1}{2}$ times less than that of Phase 1. When the new features permit appropriate generalization to occur, as in the third set of experiments, the number of errors in Phase 2 is reduced by a factor of 7 from that of Phase 1.

Learning curves for the perceptron algorithm applied to the input-cluster task are shown in Figure 2.11, and are calculated in the same fashion as those for the multiplexer task. Phase 1 is identical for all experiments: even the random number generator controlling the sequence of input vectors is the same, so the learning curves coincide throughout Phase 1. In Phase 2, the input representation is altered. The first representation used is simply the original representation, identical to the representation used during Phase 1. Not surprisingly, the learning curve in this case differs little from Phase 1 to Phase 2. The addition of standard unit basis vectors for the input classes results in a much-accelerated learning curve during Phase 2—the error is reduced to zero much faster. The representation with additional features that are equal to the desired output for each input vector results in a learning curve that is slightly better than that of the basis-vector representation, showing the effects of the beneficial generalization due to the common features of the class labels.

2.3 New Output Features—Correlations Among Output Components

Representation development is typically discussed in terms of operations applied to the input representation, as was done for the multiplexer task and the input-cluster task. Issues of new terms and new features also arise from the viewpoint of a system's output representation. One issue involves sequences of outputs, and how the time of searching for the correct output can be decreased by remembering sequences that proved useful in the past. The STRIPS (Fikes, Hart, and Nilsson, 1972) and HACKER (Sussman, 1975) systems demonstrate the utility of storing output sequences and treating the sequence as a composite, higher-level output. We do not focus on this issue, but in the experiments of Chapter VII we observe the learning in a connectionist system of short sequences of operators in a problem-solving task.

The second issue does not involve a chain of outputs, but deals with the representation of an individual output. Consider a connectionist network that has more than one output unit. The task of each output unit is to learn the desired value for a component of the output vector. For some tasks, the desired output vector is available for training the system. But when the desired output vector is not known, the system must depend on a scalar evaluation to indicate how well the output vector satisfies some criterion unknown to the system. As described in Chapter I, this kind of task can be referred to as a reinforcement-learning task, with the evaluation serving as a reinforcement value. Now the tasks faced by the output units are not independent; the reinforcement received by a unit depends not only on its output but also on the output of the other units. Output units in networks that are strictly layered have no knowledge of the behavior of other output units, so the view of a single unit is of a noisy reinforcement signal.

Some reinforcement-learning methods that successfully handle noisy reinforcements are described in Chapters III and IV. They are based on a probabilistic search over a unit's possible output values, with the probability of a particular output value determined by the unit's weighted sum of its inputs. In a single-layer system of reinforcement-learning units with no interconnections, the output of one unit has no influence on the output of others. Because the searches performed by the units before any learning takes place are independent, much time can be expended in a search for a single desired output vector when the number of possible output vectors is large. This search time can be significantly reduced by inducing correlations among the units' output values that result in a higher probability of generating one of the desired output vectors. The following experiments show how hidden units can provide these correlations through their output weights—weights connecting a unit's output to the input of other units—effectively altering the output representation of the network.

The potential gains from such methods of changing the hidden units' output weights are illustrated through the performance of a single-layer network of four reinforcement-learning units. The experiments are again composed of two phases. In the second phase, the output representation is altered to simulate the addition of hidden units with output weights that restrict the network's output to the two desired output vectors. This task is called the output-vector task. The reinforcement-learning algorithm used for these experiments is the $\mathrm{A_{R-P}}$ algorithm, defined by Barto and Anandan (1985) and also in Chapter III.

2.3.1 The Output-Vector Task

For each phase the input and the output vectors consist of four binary-valued components. In Phase 1, they are

$$x^{(1)} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad x^{(2)} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix},$$

$$d_{x^{(1)}} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad d_{x^{(2)}} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}.$$

The reinforcement signal provided to every unit on step t is labeled $r[t]$. It is a binary-valued signal whose values are "success" and "failure", and it depends probabilistically on the number of output components that are correct on step t. The probability of $r[t]$ being "success" is given by

$$P\{r[t] = \text{“success”} | x[t]; d[t]\} = 1 - \frac{1}{m} e[t],$$

where

$$e[t] = \sum_{j \in O} |d_j[t] - y_j[t]|, \quad \text{where } O = \left\{ \begin{array}{c} \text{indices of} \\ \text{output units} \end{array} \right\},$$

as defined earlier, and $m = 4$ is the number of output units. The probability of $r[t]$ being a "failure" signal is given by

$$P\{r[t] = \text{“failure”} | x[t]; d[t]\} = 1 - P\{r[t] = \text{“success”} | x[t]; d[t]\} = \frac{1}{m} e[t].$$

For Phase 2 the task was changed somewhat. Input vectors $x^{(1)}$ and $x^{(2)}$ are no longer presented. Instead, the input vectors and desired output vectors shown below are used:

$$x^{(3)} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad x^{(4)} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix},$$

$$d_{x^{(3)}} = d_{x^{(1)}}, \quad d_{x^{(4)}} = d_{x^{(2)}}.$$

The two desired output vectors for Phase 2, $d_{x^{(3)}}$ and $d_{x^{(4)}}$, are identical to those of Phase 1, $d_{x^{(1)}}$ and $d_{x^{(2)}}$. The set of input vectors are standard unit basis vectors, so no generalization can occur among them. In particular, no learning can be transferred from Phase 1 to Phase 2 on the basis of common components of the input vectors because the components present in Phase 1 are never present in Phase 2. Thus, any facilitation in Phase 2 can only be accounted for by changes in the output representation.

2.3.2 Changes to Performance Measures

The use of units with probabilistic output functions necessitates a new definition for the performance measure ν. (The cumulative error measure μ remains the same.) Let $\{a_i\}$, $i = 1, \ldots, 2^m$ be the set of possible output vectors of the system, and let a_{ij} be the component of output vector a_i produced by output unit j. The value of output component a_{ij} is called incorrect for a given input vector x if it is not equal to the component's value in the output vector corresponding to the highest probability of "success". For our simulations, this value is given by d_{xj}. The difference between d_{xj} and a_{ij} is the error in the value of a_{ij}. Although the $\mathrm{A_{R-P}}$ algorithm assumes that this error is not available and, therefore, does not use it in updating weight values, we define the performance measures in terms of this error. The expected value over all input vectors of the number of errors at the completion of run k is given by:

$$h_k = \sum_{x \in X} \left(\sum_{i=1}^{2^m} \Big(P\{y_x = a_i | x; w\} \sum_{j \in O} |d_{xj} - a_{ij}| \Big) \right),$$

where $P\{y_x = a_i | x; w\}$ is the probability of output vector a_i given input vector x and the weight vector w at the end of the run, and $\sum_{j \in O} |d_{xj} - a_{ij}|$ is the number of incorrect components of a_i for input x. As before, the number of errors is averaged over r runs to determine ν:

$$\nu = \frac{1}{r} \sum_{k=1}^{r} h_k.$$

2.3.3 Original Representation

The first experiment was designed mainly as a control experiment. The output representation of Phase 1 was maintained through Phase 2, to obtain the minimum performance level expected of a new-feature algorithm. The structure of the network during both phases is shown in Figure 2.12.

2.3.4 New Features That Correlate Output Components

The second experiment involved a change in the output representation for Phase 2. Consider the type of output representation that we might want from a new-feature algorithm operating during Phase 1. With four output components there are 2^4, or 16, possible output vectors, only two of which, $d_{x^{(1)}}$ and $d_{x^{(2)}}$, reliably result in a "success" signal. A new-feature algorithm should alter

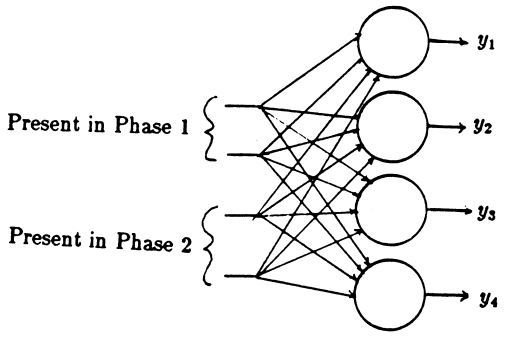


Figure 2.12: Output-Vector Task—Original Representation

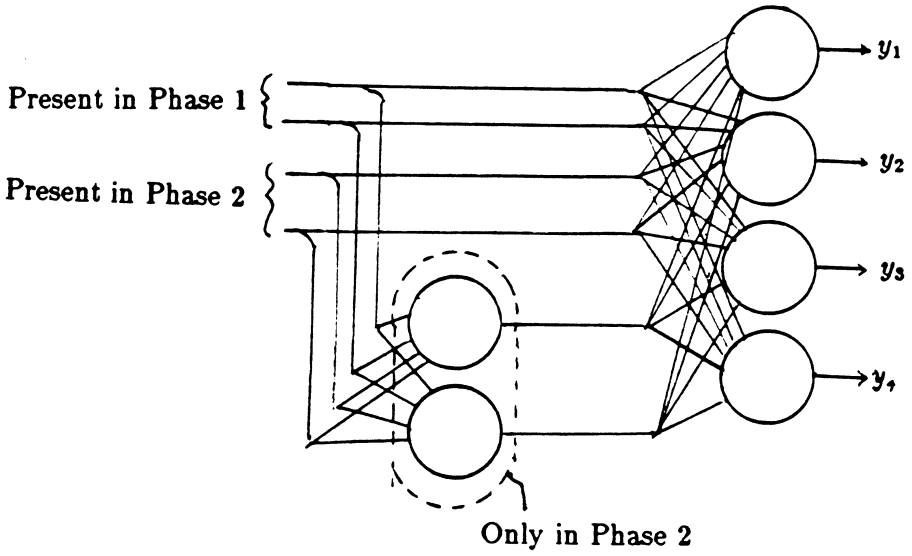


Figure 2.13: Output-Vector Task—Hidden Units That Correlate Output Units

the output representation to make $d_{x^{(1)}}$ and $d_{x^{(2)}}$ more probable than other output vectors, under the assumption that they will again be useful for obtaining the "success" signal.

This can be realized by adding two new hidden units whose outputs force the four output units to generate one or the other desired output vectors, resulting in the structure of Figure 2.13. A new-feature algorithm capable of forming such units would have little effect in Phase 1, since the task of Phase 1 must be practically solved before the statistics can be gathered that are necessary to decide which output vectors are worthy of being represented. But in Phase 2, which requires the same set of output vectors, this change would have a great impact. If the hidden units have complete control over the output units, then the set of output vectors being searched is reduced from 16 four-component vectors to a set of two two-component vectors.

To demonstrate this, we altered the output representation used during Phase 2 to one having only two output components. The input vectors and desired output vectors for Phase 2 are

$$x^{(3)} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad x^{(4)} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Representation	ν_1	ν_2	μ_1	μ_2
Original	0.04 ± 0.08	0.08 ± 0.11	220 ± 42	242 ± 41
New Features	0.02 ± 0.05	0.00 ± 0.00	201 ± 40	22 ± 5

Table 2.3: Performance of Single Layer System on Output-Vector Task

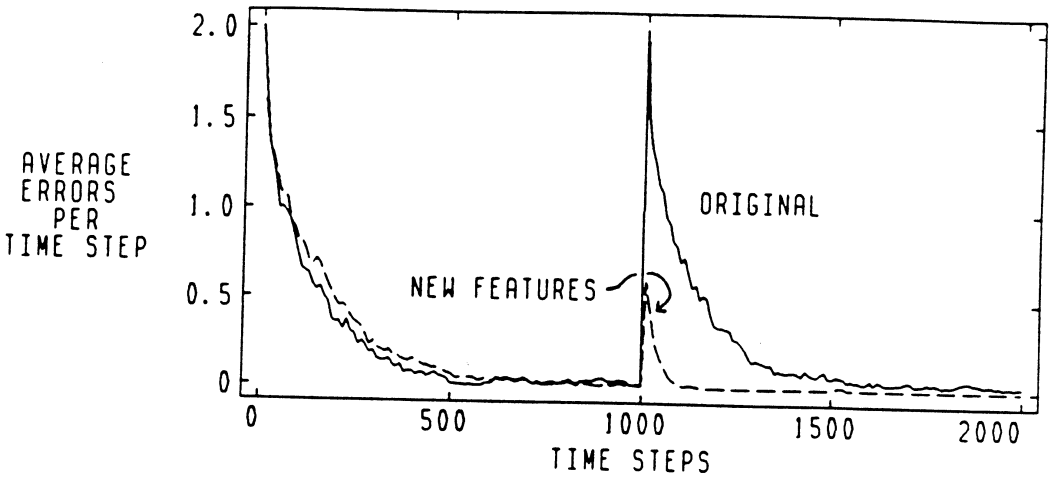


Figure 2.14: Learning Curves for the Output-Vector Task

$$d_{x^{(3)\prime}} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad d_{x^{(4)\prime}} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

2.3.5 Results

Five values for each of the parameters, ρ and λ, of the A_{R-P} algorithm were tried. For each of the 25 pairs of ρ and λ values, 100 runs were made of 2000 time steps each, with the first 1000 steps being Phase 1 and the second 1000 steps being Phase 2. Performance was measured by calculating the average number of errors at the end of each phase, given by ν_1 and ν_2, which are defined as ν was earlier with the exception that each was calculated at the end of the corresponding phase rather than only at the end of the run. Similarly, μ_1 and μ_2 are the average number of errors received throughout each phase.

The results from various values of ρ and λ are included in Appendix A. Best performance resulted when $\rho = 50$ and $\lambda = 0.1$, summarized in Table 2.3. Since there are two input vectors per phase and four output units, the maximum value of ν_i is 8. Randomly selecting the value of each output component would results in $\nu_i = 4$. The tasks of each phase are reliably solved with these parameter values.

To understand the speed with which the tasks are solved we must look at the values of μ_1 and μ_2. Each phase is 1000 steps long, so the maximum value of μ_i is 4000, and μ_i would be 2000 for a random output selection. The values of μ_i reflect that no facilitation occurs from Phase 1 to Phase 2 with the original representation in both phases. However, the "simulated" development of new features that select one or the other desired output vector results in a 9-fold decrease in the number of errors accumulated during Phase 2 (μ_2) as compared to Phase 1 (μ_1).

Learning curves are shown in Figure 2.14. The learning curves of 100 runs are averaged into

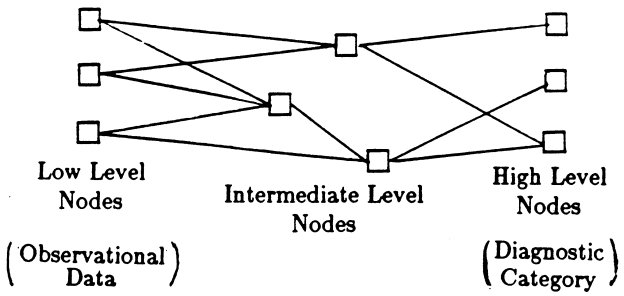


Figure 2.15: Multi-Level Reasoning Network (from Fu and Buchanan [1985])

intervals 10 steps in length. With the original representation used for both phases, there is no difference in performance. With the new output representation in Phase 2, a large decrease in the amount of time needed to reach a solution is seen during Phase 2. The difference in the Phase 1 learning curves is due only to the stochastic output function of the A_{R-P} algorithm.

2.4 New Intermediate Concepts in Production Systems

The previous two sections illustrate two ways in which hidden units can facilitate learning in connectionist systems: new features can be formed that have similar values for a set of input vectors requiring the same output, and output units can be constrained to produce desirable output vectors. Similar ideas have been proposed by Fu (1985; Fu and Buchanan 1985) for learning intermediate concepts in multilayered production systems. Their work is briefly summarized in this section, and the relationships between the learning of intermediate concepts in production systems and the learning of new features in connectionist systems are discussed.

Fu and Buchanan describe the processing performed by a rule-based expert system by means of a *multi-level reasoning network*, an example of which is shown in Figure 2.15. Observational data define the low level nodes in this network, from which the knowledge base of rules ultimately draws conclusions concerning high level concepts, such as diagnostic categories if the system is performing a medical diagnostic task. In reasoning from low level to high level nodes, *intermediate concepts* like generalized findings (observational data) and generalized disease classes might be generated. Fu and Buchanan state the following advantages of a rule base that reasons through multiple levels, as opposed to rules that directly relate observational data and high level concepts:

1. Reasoning proceeds in smaller steps.
2. Explanations of the reasoning process is more understandable.
3. Resulting generalization is useful when insufficient data is available.

Comparisons between the multi-level reasoning network implicitly defined by the rule base and the explicit multilayer network of a connectionist system can be made by considering both "concepts" and "features" as names for partitions of a system's input space. In both systems, processing spreads from the observational data to a final decision regarding the output of the system, guided by the constraints imposed either by rules or by numerically-weighted connections. Further comparisons are made below regarding learning.

Fu and Buchanan address the problem of learning *intermediate level knowledge*—intermediate concepts and relationships among intermediate concepts and concepts on other levels—when presented instances described only by low level data and high level concepts. They distinguish the methods for learning the relationships when intermediate concepts already exist from methods for learning new intermediate concepts. A similar distinction cannot always be made in the learning methods for connectionist systems; the modification of features and the adjustment

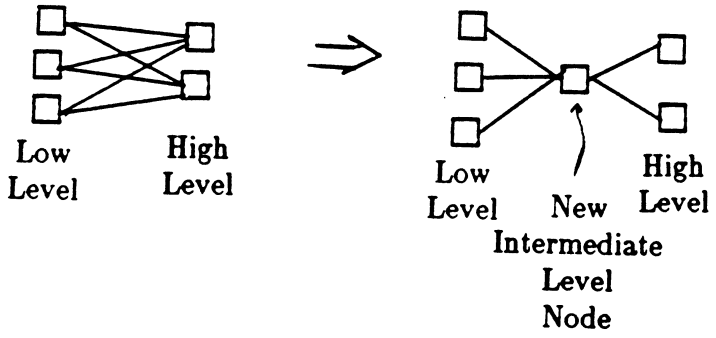


Figure 2.16: Naming Switchover Points

of their influence on other units are usually performed simultaneously by the same mechanism operating on different weights.

Here we focus on the methods for learning new intermediate concepts, since this is most relevant to the problem of learning new features in connectionist systems. Fu and Buchanan describe two approaches to the learning of intermediate concepts, called the techniques of naming *taxonomy points* and *switchover points*. Each is discussed in turn below.

2.4.1 Taxonomy Points

This method proceeds as follows. A taxonomy tree of the instances is constructed, according to a similarity measure between classes of observational data for each high level concept. High level concepts found to have similar conditions on the observations become linked through a taxonomy point, which is given a name and becomes a new intermediate concept. The links between high level concepts and new intermediate concepts drive the process of learning new rules relating these new intermediate concepts with others.

The process of naming taxonomy points is akin to the method of inducing correlations among the output units of a connectionist network as illustrated by the experiments with the output-vector task. Viewing the values of the output units as high level decisions, the output vector for every step is seen as a set of high level decisions. If the input conditions under which particular output values of two or more output units are similar, then these output values will co-occur. A learning algorithm designed to recognize these co-occurrences can adjust the output weights of hidden units to give the units the ability to generate these patterns of output values. Such hidden units become explicit analogs of Fu and Buchanan's taxonomy points. Once they are formed (or while they are being formed), other connection weights can be adjusted to learn relationships between these hidden units and observational data, and other features. The end results of this connectionist learning process and the naming of taxonomy points are very similar, though the methods differ. In naming taxonomy points, the observational data in the set of instances plays a key role in determining the similarity of high level concepts. In contrast, the proposed connectionist method relies on the temporal co-occurrence of high level decisions to indicate their similarity.

2.4.2 Switchover Points

A switchover point is formed by funneling the links from a set of low level nodes that directly connect to identical high level nodes through a single new intermediate level node. This process is illustrated in Figure 2.16. Fu and Buchanan point out that most of the intermediate concepts that were formed in this way for a medical diagnostic task are medically meaningful because switchover points only exist between low and high level nodes that have very regular relationships.

A direct comparison of this situation to the input-cluster task can be made. The input-cluster task involved classes of input vectors where the members of each class were associated with identical output vectors. This is exactly the relationship shown in Figure 2.16. In the connectionist system experiments, a new feature was created that had the value 1 for all input vectors in a class, and the value 0 for all other input vectors. If this feature is learned by a hidden unit, then the output of that unit can be linked to output units so as to generate the output vector appropriate for that input class. Thus, the hidden unit plays the role of the switchover point.

One aspect of Fu and Buchanan's work sets it apart from Utgoff's work in the same way that the multiplexer task is different from the other two tasks of this chapter. This is the fact that Utgoff devised ways of learning new terms that were necessary for his system to form a solution, just as new features were necessary for the two-layer connectionist system applied to the multiplexer task. On the other hand, Fu and Buchanan start with a solution to a task in terms of rules directly relating low and high level concepts, and wish to transform this single-layer solution to one having multiple levels of intermediate concepts in order to achieve better generalization and to facilitate explanations. Similarly, the input-cluster and output-vector tasks are were to illustrate how new features lead to better generalization for tasks that can be solved by a single-layer system. It is also possible that new features, viewed as intermediate level concepts, will prove useful in generating an explanation of the processing performed by a connectionist system in determining which output vector to generate. Gallant (1985) has made some initial steps along these lines.

2.5 Summary

The experiments of this chapter demonstrate several kinds of new features and how they facilitate learning in connectionist systems. The benefits of learning missing features was illustrated by means of the multiplexer task, which is used in Chapter IV to compare the abilities of several learning algorithms to find missing features. The other two tasks do not appear again in this thesis, although the issues they expose are no less important than the issue of missing terms. We chose to study the issue of missing features and how they can be learned for the multiplexer task, and for two strategy learning tasks. Similar treatments of the other ways in which new features facilitate learning remain to be performed.

Before presenting further multiplexer experiments, past and recent approaches to learning in the hidden units of connectionist systems are reviewed. A very important observation is that most, if not all, learning methods for multilayer systems are motivated by the need to learn missing features, and do not deal with learning features that are not needed but that do facilitate learning. Such methods must continue to learn even after a task is solved in order to develop additional features, or intermediate-level concepts, that might be useful for further learning in the future.

Chapter 3

Review of Learning Methods for Hidden Units

This review assembles in a consistent form both early and recent work on learning in the hidden units of connectionist systems. In addition, numerical methods not originally presented as being used in connectionist systems but that can be cast as connectionist learning methods are included. The goal of this review is to provide an opportunity to compare and contrast the various learning methods and to indicate the types of tasks for which they are applicable. Some of the methods reviewed here are compared by their ability to learn new features in the experiments of Chapter IV.

3.1 The Connectionist Learning Problem

Consider the three-layer connectionist network shown in Figure 3.1. Let us say that Unit i calculates an output value by applying a function, y_i, on a weighted sum of its input values. The final output of the network, labeled $F_w(x)$, is a composite of the output functions of every unit. For the network shown, this composite function is

$$\begin{aligned} F_w(x) \;=\; & y_4\Big(w_{1,4}\, y_2\Big(w_{1,2}\, y_1(w_{1,1}x_1 + w_{2,1}x_2) + w_{2,2}x_2 + w_{3,2}x_3\Big) \\ & + w_{2,4}\, y_3(w_{1,3}x_1 + w_{2,3}x_2 + w_{3,3}x_3)\Big). \end{aligned}$$

Of course, for larger networks the complexity of this expression increases: additional layers result in additional levels of nested applications of y_i and additional units in a layer result in more y_i's at one level of nesting. The output of a network is thus a complex function parameterized by the vector of interconnection weights, w, represented by the subscript of $F_w(x)$. The problem of learning a particular mapping from input x to output $F_w(x)$ is one of determining a set of parameter values, w, that result in as close an approximation to the correct mapping as desired.

Methods of searching for good weight values vary in the amount of knowledge assumed about the structure of the network and about the goal of the learning task. When such knowledge is unknown, *direct search* methods (Gill, Murray, and Wright, 1981) must be applied that are based only on comparisons of network performance resulting from the use of different weight values. The entire weight space of the network is searched in an attempt to optimize the network's performance. Several direct search methods are reviewed below, after which ways of incorporating structural and goal knowledge are discussed.

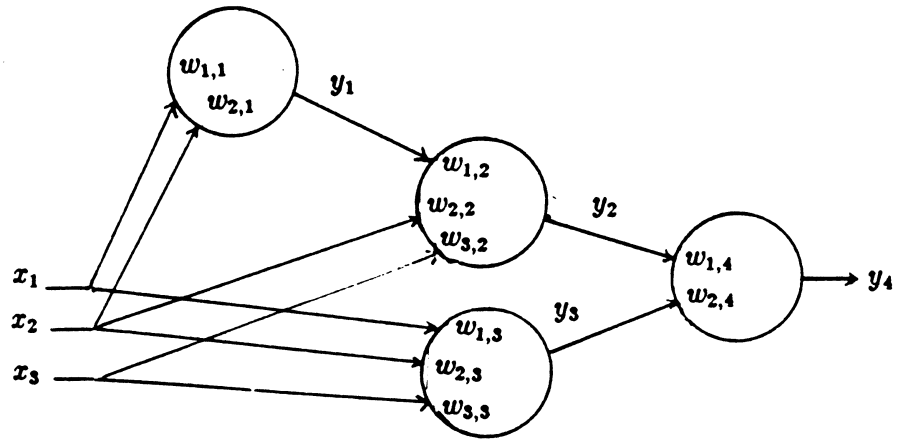


Figure 3.1: A Connectionist Network

3.2 Direct Search

Each weight vector can be considered a point in the network's weight space. The fundamental ingredient of a direct search method is the choice of which point to try next. The decision can be deterministic or random. A second distinction is the degree to which previously-tested points bear on the decision.

The *polytope* algorithm (Gill, Murray, and Wright, 1981) is a deterministic procedure for determining a new point based on the n best previously-tested points, where n is a parameter. These points are considered as the vertices of a geometric figure—a polytope in $n-1$ space. The centroid is calculated by averaging the coordinates of the vertices, and the vector from the worst point to the centroid is taken as a plausible direction of increasingly-good points. A new point is generated by "reflecting" the worst of the n points through the centroid to the opposite side of the polytope. Further calculations are performed depending on the outcome of testing the new point. The polytope algorithm is defined in detail in Chapter IV.

The polytope algorithm is a "hill-climbing" method, in that it takes small steps in a direction that is assumed to be towards points that result in better performance. Hill-climbing methods can become stuck at local minima (or maxima, if trying to maximize a criterion), so they are generally not recommended for optimization tasks having multimodal criteria, i.e., having local minima (or maxima) in the criterion function.

A very straightforward approach which is not a hill-climbing search is to simply pick points in the weight space at random, with every point having equal probability of being selected. The point that results in the best performance is remembered. This *unguided* search (labeled by Gilstrap, 1971) is not trapped at local minima, but would take an impractical amount of time to find a minimum for large weight spaces.

A compromise between the unguided search and the deterministic hill-climbing search of the polytope algorithm can be achieved in a number of ways. One way is to base the choice of new points on a probability distribution, such as a Gaussian distribution, centered on the currently-best point. New points tend to be on the same hill as the currently-best point, but there is a nonzero probability that a point on a different hill will be sampled. The selection of the new point can be given a bit more direction by restricting new points to lie within a region bounded by a cone, with its axis parallel to the line joining the two best points and its vertex at one of the points (Yudin, 1966; see Jarvis, 1975, for a review of both random and deterministic methods). This is a *guided* random search.

Rastrigin (1963) analyzed a guided random search technique and showed that it converged faster than gradient methods (described below) for systems having a large number of parameters. Bekey and Masri (1983) provide results from the application of a guided random search to several

identification tasks. However, the space of possible weight values is so large for all but trivial networks of a small number units that a direct search of the network's weight space is very slow to converge upon a good set of weights. The results of Chapter IV show this to be true for a two-layer, five-unit network to which an unguided random search, a guided random search, and the polytope algorithm were applied.

To reduce the search time, knowledge of the network's structure must be used to constrain the search. Often this entails the formulation of a search at the unit level, i.e., a search for good units. The weights of a unit are modified in a consistent fashion, so as to either increase or decrease the output value of the unit for the given input. The introduction of structural knowledge to the search for good weights gives rise to the issues of:

- the assignment of credit to the units and weights of a network, and
- the modification of weights based on assigned credit.

These issues are described below, and ways with which they have been addressed are reviewed.

3.3 Assignment of Credit

As discussed in Chapter I, the credit-assignment problem has several forms. The assignment of credit to the units of a connectionist network is a form of *structural* credit assignment. "Credit" is a broad term encompassing several kinds of information that can be presented to the units to guide the adjusting of weights. The methods reviewed here assign credit in one of the following forms:

1. the gradient of a criterion function with respect to a unit's weights,
2. an estimate of the gradient,
3. an error based on a minimal change,
4. the worth, or usefulness, of a unit.

The first three forms of credit result in the assignment of errors to the weights, either by the calculation or approximation of a gradient or by restricting changes to be of minimal size. For these cases, credit not only indicates which weights should be adjusted but also how they should be adjusted to increase the network's performance. The fourth method does not include this information—worth is only an evaluation, not an instruction as to how weights should be modified. The following review of credit-assignment methods is organized according to the above list.

3.3.1 Exact Gradient Methods

A common optimization technique is to calculate the gradient of a criterion function with respect to the parameters of the function to be optimized. Gradient-descent procedures can then be applied to shift the parameter vector towards a local minimum in the criterion function. Gradient-descent procedures perform poorly when the first and second derivatives of the criterion function do not vary smoothly. For example, they do not constrain the search to be along the floor of a ravine in the criterion function. Ways of dealing with this issue are described below.

Rumelhart, Hinton, Williams, 1986

If the criterion function to be minimized by a connectionist learning method is denoted $J(w)$, then a straightforward gradient-descent technique can be applied if $J(w)$ is differentiable with respect to each $w_{i,j}$. Assuming that the criterion is a function of the network's output, $F_w(x)$, this implies that each y_i is differentiable with respect to Unit i's weights. Rumelhart, et al., derived

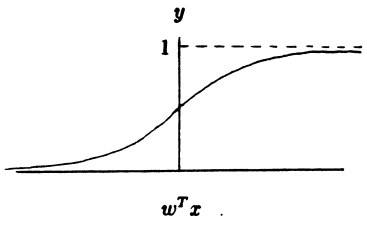


Figure 3.2: The Logistic Distribution

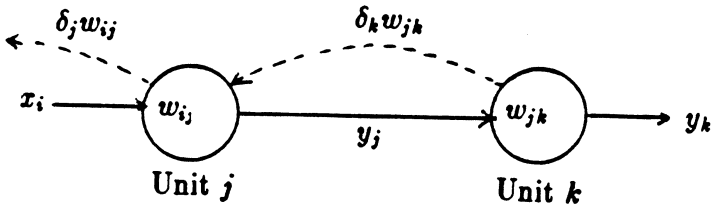


Figure 3.3: Back-Propagation of Gradient Information

a gradient-descent technique for updating the weights of a network by using the squared-error criterion:

$$J(w) = \sum_{x \in X} (d_x - F_w(x))^2$$

and the differentiable output function

$$y_i = \frac{1}{1 + e^{-w^T x}},$$

where w and x are the weight vector and input vector for Unit i and $w^T x$ is the inner product of these vectors. The shape of y is shown in Figure 3.2. It is a "smoothed" version of the standard linear threshold unit—when the value of $w^T x$ varies widely for different x, y approximates the discontinuous response of the linear threshold unit. (This discontinuity prevents the use of gradient techniques for linear threshold units.)

Rumelhart, et al., remove the centralized nature of gradient-descent techniques by casting the gradient-descent expressions as weight-update rules that use information available locally to each unit plus a quantity that is *back-propagated* from other units. Consider a Unit j whose output is connected to a number of other units with indices in set K, as pictured in Figure 3.3. Let the back-propagated value from Unit k to Unit j be the product $\delta_k w_{jk}$. The value of δ_j for any Unit j is

$$\delta_j = \begin{cases} \Big(d_j - y_j\Big)\, y_j \Big(1 - y_j\Big), & \text{if } j \in O; \\ \displaystyle\sum_{k \in K} \Big(\delta_k\, w_{jk}\Big) y_j \Big(1 - y_j\Big), & \text{otherwise,} \end{cases}$$

where O is the set of indices for the output units, and d_j is the desired output value for Output Unit j. This quantity is used to update the weights of Unit j according to

$$\Delta w_{ij} = \rho\, \delta_j\, x_i,$$

before the unit back-propagates $\delta_j w_{ij}$ to Unit i. Since this algorithm is deterministic, a random process is used to initialize the weights to small random values—otherwise the units in a hidden layer would evolve identical weights.

At least two modifications to this update rule have been proposed for dealing with the problem of searching ravines mentioned above. Rumelhart, et al., add a term that they refer to as "momentum", resulting in the following update rule:

$$\Delta w_{ij}[t+1] = \rho\, \delta_j\, x_i + \rho_m\, \Delta w_{ij}[t],$$

where ρ_m, which is greater than zero, controls the amount of past weight changes that are added to the current weight change. This has the effect of increasing the step size when the gradient does not change sign, so a smaller value of ρ can be used to limit cross-ravine jumps. The momentum of each weight is distinct, enabling large changes in some weights while maintaining small changes in others. This has the desired effect for ravines that are oriented parallel to a dimension of weight space, i.e., a change in the magnitude of weight changes constrain the search to weight values near the floor of the ravine.

A second modification is to only use the sign of the weight w_{jk} in the back-propagated quantity from Unit k to Unit j. This has a normalizing effect of reducing the gradient with respect to units with large output weights—weights connecting a unit's output to the input of another unit—while increasing the gradient with respect to units having small output weights (Sutton, 1985, 1986).

3.3.2 Approximate Gradient Methods

When differentiable output functions are not used, exact gradient methods cannot be applied. Rosenblatt was one of the first to heuristically design a procedure for estimating the gradient of a criterion with respect to the weights of hidden units. He called his method the *back-propagating error correction procedure* (Rosenblatt, 1962). It is similar to Rumelhart's algorithm in that an error is back-propagated, but it differs by using a nonanalytic, probabilistic determination of errors.

Rosenblatt, 1962

Rosenblatt's procedure for two-layer networks is as follows. Referring to Figure 3.3, let $K = O$, i.e., k is an index to an output unit. The error for an output unit is calculated as

$$\delta_k = d_k - y_k,$$

where d_k is the desired output of Unit k, and y_k is its actual output. If $\delta_k = 0$ for all k, then no errors are back-propagated and no weight change is made. When $\delta_k \neq 0$, the error that is back-propagated from Unit k to Unit j, called δ_{jk}, is determined by one of the following cases:

- If $y_j = 1$ and δ_k differs in sign from w_{jk}, then $\delta_{jk} = -1$ with probability p_1.
- If $y_j = 0$ and δ_k agrees in sign with w_{jk}, then $\delta_{jk} = 1$ with probability p_2.
- If $y_j = 0$ and δ_k differs in sign from w_{jk}, or $w_{jk} = 0$, then $\delta_{jk} = 1$ with probability p_3.

The hidden unit weights are modified by

$$\Delta w_{ij} = \rho\, \mathrm{sgn}\left(\sum_{k \in O} \delta_{jk}\right)\, x_i.$$

Rosenblatt proved a theorem stating that if a solution exists, it can be found by this procedure in finite time with probability 1. His simulations show that the procedure is very sensitive to the parameters, p_1, p_2, and p_3; for some cases certain parameter values resulted in lower performance than that achieved by a system with fixed hidden unit weights. He concluded that instabilities are apt to arise due to changes in one layer before the other layer is able to converge on a good set of weights. Instabilities can be reduced somewhat by adjusting the output-unit weights first, and only back-propagating errors if the output-unit errors persist.

Alder, 1975

A simpler way to assign errors was analyzed by Alder (1975), who developed a learning algorithm for multilayer systems of perceptron-like units having a single output unit. Alder's algorithm for picking a unit to be modified proceeds as follows. If the output of the system is wrong, then a unit is chosen randomly from the entire system (including the output unit) with a uniform probability. The error assigned to the output unit is given to the randomly chosen unit and Rosenblatt's perceptron learning rule is applied to update its weights. If the output of the system is correct, then no units are adjusted unless the previous output was wrong, in which case a unit is chosen randomly from a set consisting of all units plus an element that signifies no change, and the perceptron learning rule and the output unit's error are applied as before. In other words, an incorrect output followed by a correct output leads to the modification of a unit (chosen randomly) in $n/n+1$ cases, where n is the number of units in the system.

Alder presented a variation of the perceptron convergence theorem for his algorithm, for which he proved that the probability is zero that the system would *not* converge in some finite number of steps. He points out, though, that this is a probabilistic convergence proof, and that the algorithm is "less than efficient." Convergence speed can be increased for certain network architectures by restricting corrections to units that are wrong, but in general one cannot determine whether a unit is wrong or right.

An alternative to constructing an ad hoc gradient estimate, as Rosenblatt did, or to directly assigning the output error to hidden units, as Alder did, is to add noise to the output of the units and accumulate statistics with which a gradient can be approximated. Descriptions of several techniques for performing a stochastic search follow.

Ackley, Hinton, Sejnowski, 1985

By using units with stochastic output functions and symmetric connections between units, Ackley, et al. (1985), derived an expression for updating each weight based only on information about the pair of units associated with the weight.

A *Boltzmann Machine*, as they refer to their system (Hinton, Sejnowski, and Ackley, 1984), consists of units whose outputs are either 1 or 0, determined probabilistically by the weighted sum of their inputs, $w^T x$, and by the following expression, known as the logistic function, for the probability of a unit's output being equal to 1:

$$P\{y=1\} \;=\; \frac{1}{1+e^{-\frac{w^T x}{T}}} \qquad \text{and} \quad P\{y=0\} \;=\; 1-P\{y=1\},$$

where T is a "temperature" parameter that scales the weighted sum, effectively controlling the amount of noise in a unit's output. Drawing from work in modeling the thermodynamics of two-state particles (Binder, 1978) and on constraint satisfaction (Kirkpatrick, Gelatt, and Vecchi, 1983), Ackley, et al. showed how a process of "simulated annealing"—a gradual lowering of the temperature T—can be used to search for output values that closely match the constraints imposed by the interconnection weights. The process is repeated to obtain statistically significant estimates of the probability of each output vector at low temperature values.

Visible units are units whose output can be observed or controlled by the trainer. It is the probabilities of visible-unit output vectors that are to match the desired probabilities. Repetitions of simulated annealing are performed in a "clamped" mode and an "unclamped" mode, referring to whether or not the outputs of the visible units are fixed, i.e., clamped to the desired values. Let $P\{V_\alpha\}$ be the probability of V_α, a vector of visible-unit output values at thermal equilibrium arrived at by simulated annealing in clamped mode, and let $P'\{V_\alpha\}$ be the probability of the same vector in unclamped mode. A measure of discrepancy between the two is defined as

$$G = \sum_\alpha P\{V_\alpha\} \ln \frac{P\{V_\alpha\}}{P'\{V_\alpha\}}.$$

The partial derivative of G with respect to weight w_{ij} is

$$\frac{\partial G}{\partial w_{ij}} = -\frac{1}{T}(p_{ij} - p'_{ij}),$$

where p_{ij} is the average probability of the outputs of units i and j both being 1 when the visible units are clamped, and p'_{ij} is the probability when the visible units are unclamped. Using the partial derivative of G in a gradient-descent rule for updating weight w_{ij} results in

$$\Delta w_{ij} = w_{ij} + \rho\,(p_{ij} - p'_{ij}).$$

A variant of the above update rule was used by Ackley, et al., to deal with the difficulties posed by ravines in the criterion function G. The $(p_{ij} - p'_{ij})$ term was replaced by $\mathrm{sgn}(p_{ij} - p'_{ij})$, using only the direction of the gradient and disregarding its magnitude.

The Boltzmann Machine learning algorithm relies on accurate estimates of output probabilities at thermal equilibrium, requiring many time-consuming stochastic searches. Rather than waiting for an accurate estimate of the gradient as in the Boltzmann Machine paradigm, information from each step in a stochastic search can be applied to the update of the weights. In a sense, the stochastic search is shifted to the weight space, allowing a quicker benefit to be gained by the modification of output probabilities through weight adjustments on each training step. To guide each weight adjustment, a signal must be available that evaluates the current weight values. The error back-propagation schemes of Rumelhart, et al., and Rosenblatt do provide information on each step, but are based on deterministic output functions. Without the assumptions of differentiable output functions or of particular error-assignment heuristics, we need some way of combining stochastic search with the evaluative information available on each step. Due to the similarity of this approach to mathematical models of animal learning, the evaluation signals are referred to as *reinforcement* signals, which increase the future probabilities of the animal's actions that lead to desirable outcomes.[1]

Minsky, 1954

Minsky investigated a type of reinforcement-learning in his Ph.D. thesis (Minsky, 1954), where he described the SNARC (Stochastic Neural-Analog Reinforcement Calculator). The units of his system are simple channels with single weights determining the probability of passing a pulse through the channel. The probability is increased if the passing of a pulse is followed by a reward. Minsky demonstrated his system in a maze-learning task.

Farley and Clark, 1954

Another approach to reinforcement learning was developed by Farley and Clark (1954), who used output functions approximating the observed response of neurons: a noisy threshold was employed to produce a probabilistic, binary output. Networks of units consist of randomly placed connections and weight values. Weights are modified according to the simple rule of increasing a weight if a) the unit preceding the weight in the network "fired", b) the following unit's output increased, and c) the system's evaluation increased. If the evaluation decreased, the weight is decreased. The units are modified under a high degree of uncertainty—the probability of a particular unit having a significant effect on the overall evaluation is very low for networks of many units.

Barto, et al., 1981–present

Several types of reinforcement-learning algorithms similar to that of Farley and Clark have been studied by Barto, et al. (Barto, 1985; Barto and Sutton, 1981a; Barto, Sutton, and Anderson, 1983; Barto, Sutton, and Brouwer, 1981; Sutton, 1984), by developing models of animal learning

[1] This is a simplified statement of the "Law of Effect" of animal learning theory (Thorndike, 1911).

(Sutton and Barto, 1981) and Klopf's (1972, 1982) theories of learning in neural networks. These algorithms have been successfully applied to learning in small two-layer networks (Anderson, 1982; Barto, 1985; Barto and Anderson, 1985; Barto, Anderson, and Sutton, 1982).

Recently, Barto and Anandan (1985) devised a new class of reinforcement-learning algorithms that is an extension of the Linear Reward-Penalty ($\mathrm{L_{R-P}}$) algorithm from the learning automata literature (Narendra and Thathachar, 1974). The $\mathrm{L_{R-P}}$ algorithm learns probabilitities for selecting actions that maximize the probability of receiving a good reinforcement. Barto and Anandan's extension is the use of the weighted sum of a unit's input to calculate the probability of producing a binary-valued output, thus allowing different action probabilities to be learned for different input vectors. Their algorithm is called the Associative Reward-Penalty ($\mathrm{A_{R-P}}$) algorithm, because it can associate different output probabilities with each input.

The probability of the output of an $\mathrm{A_{R-P}}$ unit being 1 can be given by the logistic function, as in the Boltzmann Machine, though the $\mathrm{A_{R-P}}$ algorithm is not restricted to this function. The reinforcement signal is assumed to be binary-valued, being either a "reward" or "penalty" symbol (a real-valued reinforcement version also exists). The meaning of these symbols is not important to the functioning of the algorithm, except that the probability of reward must be greater than the probability of penalty when the output of the network is the preferred value for a given input. Weights are updated according to the following rule:

$$\Delta w_{ij} = \begin{cases} \rho\Big(y_j - E\{y_j|w;x\}\Big)x_i, & \text{if reward;} \\ \lambda\,\rho\Big(1 - y_j - E\{y_j|w;x\}\Big)x_i, & \text{if penalty,} \end{cases}$$

where $E\{y_j|w;x\}$ is the expected value of y_j given Unit j's current weights and input, and $\rho > 0$ and $0 < \lambda \leq 1$. Barto and Anandan proved that this algorithm will converge to solution weights if they exist and if the input vectors are linearly independent. The $\mathrm{A_{R-P}}$ algorithm has been demonstrated in two-layer networks where a single reinforcement signal is provided to all units, i.e., all units receive the same global evaluation signal rather than unique signals (Barto, 1985; Barto and Anderson, 1985).

When $\lambda = 0$, this algorithm is called the Associative Reward-Inaction ($\mathrm{A_{R-I}}$, after the related $\mathrm{L_{R-I}}$ algorithm for learning automata) because weights are not changed upon receipt of penalty. Williams (1986) shows that the $\mathrm{A_{R-I}}$ algorithm with the logistic output function results in expected weight changes equal to the gradient of the expected value of the reinforcement signal. The penalty term of the $\mathrm{A_{R-P}}$ algorithm appears to push the search for good weights away from local minima, though the effect of the penalty term is not fully understood.

3.3.3 Minimal Change

Gradient methods, whether exact or approximate, provide an expression for the assignment of error to a unit and its weights. When gradients are not used, some other principle for the assignment of errors must be applied. In this section the principle of *minimal change* is discussed, whereby errors are assigned by considering the amount of change required to remove the errors.

Widrow, 1962

In his work with networks of "Adalines" (for adaptive, linear elements), Widrow developed the following method for learning in two-layer networks. First, the weights of the output units are examined to decide which set of hidden units can be altered to remove the output error. If several such sets of hidden units exist, then the set with the fewest units is chosen, and if there is more than one minimal set with the same number of units, the set requiring the least total change in magnitude of the weights is selected. If no combination of changes in the hidden units would remove the output unit error, then the output units are adapted and the hidden units are left unchanged. Widrow referred to this procedure as a way of "load-sharing" the training among the hidden units, in that "responsibility is assigned to the units that can most easily assume it." Tests with two-layer networks of only three hidden units and a single output unit were reported. The difficulty of determining sets of hidden units whose adjustment can remove the errors increases considerably with more output units and more layers.

Stafford, 1963

Another way to select the units requiring the least amount of change was described by Stafford. His method is applicable to systems of arbitrary structure, with the restriction of only one output unit. Linear threshold units are used whose outputs are either $+1$ or -1, with the addition of an unweighted "bias" input that has the same value for all units. Weights are restricted to positive values in order to decrease the difficulty of determining the correct response for units whose outputs are transformed by many layers before influencing the output unit. Thus, if the output of the output unit is -1 and should be 1, the output of some hidden unit, or units, must be changed from -1 to $+1$. The trick, as always, is to determine which units to alter.

Stafford's solution is to manipulate the external bias signal as follows. Consider a particular situation in which the system's output should be $+1$, but is actually -1. For this case, the bias is increased until the output changes to $+1$, and then the bias is slowly decreased until the output reverts to -1. At this point, units whose output values have just changed from $+1$ to -1 change their weights to make their output values $+1$ again. The output of the output unit is again correct, and the bias continues to decrease. This process is repeated until the bias reaches zero or becomes slightly negative. If the desired output is -1 rather than $+1$, the bias and weight changes are reversed. Stafford claimed that his solution approximated three goals:

- units selected for changing should be those whose output would also be changed for a minimum number of other possible situations,
- changes in the selected units should contribute significantly towards correcting the system's output, and
- a minimum number of units should be changed.

In subsequent work, Stafford (1965) developed a second method for which the restriction of weights to positive values is dropped, requiring a more involved procedure for perturbing the output of the units than provided by the single external bias value. A non-negative bias signal is used and each unit randomly chooses the sign with which the bias is added to the unit's weighted sum. The bias's magnitude starts at zero and is slowly increased while the units randomly vary the signs of their biases. When the output of the network becomes correct, the current signs with which the bias affects the units are frozen, and the magnitude of the bias is slowly decreased to zero while Stafford's first method is used to adapt the weights.

Another method for finding the units requiring the minimal amount of change to correct an error is through the use of *prototype* units. A prototype unit generates the largest output value for an input vector that matches the unit's weight vector, which defines the prototype, and the unit generates lower values as the input becomes increasingly different from the prototype. Typically, the output is restricted to be nonnegative and thus becomes zero for inputs a certain distance (in the input space) from the prototype.

Soklic, 1982

Soklic developed a model for decision making that was presented as a production system that can be equivalently cast in terms of a two-layer connectionist system as follows. The weights of each hidden unit determine a rectangular domain in the input space, with the edges of the domain parallel to the coordinate axes of the space. The output function of a unit produces a constant, nonzero value for inputs falling inside the unit's domain and zero for all other inputs. The output layer simply associates a classification with each hidden unit. Thus, the input space is divided into possibly overlapping, rectangular domains of various sizes with a classification assigned to each.

Of interest here is the method employed to alter the size and location of the domains. A very small number of units must be considered for modification, since the only units of interest are those with domains that encompass the current input. To begin each iteration of the training phase, an input from the training set is presented and the system's output is calculated. If the output is not the correct classification, then a change is made in the weights of the hidden units as follows.

First an attempt is made to correct the error by modifying hidden units to include the current input in a domain associated with the correct classification. An existing domain is expanded to include a new input if:

1. the domain is associated with the correct classification,
2. the increase in its volume (scaled according to the number of past inputs falling within the domain) will not exceed a fixed amount, and
3. the increased domain will not overlap "too much"[2] with other domains that are associated with incorrect classifications.

When the above conditions are not met, a new domain, i.e., a new hidden unit, is added. If the input is not "too close" to existing domains, the new domain is centered on the input and is given sides of equal length. The initial width of a new domain increases with each additional domain. If the input is "too close" to an existing domain, then the new domain, again centered on the input, is given the shape of the closest domain, except for edges that exceed a certain length on the assumption that domains in neighboring parts of the input space should be of similar shape.

Soklic applied his system to a medical diagnosis task and concluded that it performed slightly better than the performance of three specialists and much better than three internists on a set of 82 cases.

Reilly, Cooper, and Elbaum, 1982

By defining units whose outputs are nonzero for rectangular domains, Soklic minimized the difficulty of determining which units could most easily be changed without disrupting the system's output for other inputs. Similarly, Reilly, et al., adopted a geometric view of the input space in defining the domains of hidden units. However, rather than explicitly defining the units' functions to produce such domains, they employed linear threshold units and normalized the set of possible input vectors. The weights of a hidden unit thus define a prototype: the output of a unit is nonzero for a circular domain on the unit hypersphere defining the space of normalized input vectors, and beyond the circular domain the unit's output is zero. Associated with each hidden unit is a size factor, which scales the prototype before the threshold is applied, thus determining the size of the circular domain. The output layer associates the output of the hidden units with a classification.

Reilly, et al., described two steps in which learning occurs. The first step arises when the system does not classify the current input in the correct category. A new hidden unit is added and its prototype is set equal to the current input. Its size factor is set to some initial value and its output is associated with the correct classification. In addition, if the input is classified as a member of an incorrect category, a second step is performed. The size factors of nonzero hidden units that are associated with the incorrect category are reduced to the point where the outputs of the units become zero. Note that the size of the domains only decreases, as opposed to the increases and decreases in size that Soklic's system performs.

Hampson, 1983

The normalization of input vectors allows linear threshold units to divide the input space into circular domains of varying size. Another restriction on the input representation was proposed by Hampson for similar purposes. His two-layer system was designed for inputs represented by binary vectors. Linear threshold units were used whose outputs ranged from -1 to 1. The units are said to perform classifications by prototypes, since each unit can be tuned to indicate the presence or absence of a specific set of binary components.

One learning stage, called "goal-driven processing", assumes that hidden units are responsible for the errors in the output layer. During this stage, hidden units whose output is $+1$ when an error occurs are "focused" by shifting their weight vectors toward the current input vector, and their discrimination is increased by reducing their output values. Any areas of the input space

[2]The meaning of "too much" and "too close" are defined by Soklic.

that are no longer represented by a +1 output of at least one hidden unit are expected to become represented by new units through the "input-driven categorization" stage.

Hampson's "input-driven categorization" process comes to represent input vectors through a competition process among the hidden units (Rumelhart and Zipser, 1985, discuss a similar approach). Let us denote the weighted sum of hidden unit j as s_j and the maximum and minimum values of the weighted sums over all hidden units as s_{max} and s_{min}, respectively. When an input vector is presented for which no hidden unit responds with a nonzero value, all hidden units are given a desired output value of 1 and the weights of hidden unit j are adjusted by amounts proportional to $(s_j - s_{min})/(s_{max} - s_{min})$. In this way, the unit that requires the smallest weight changes will come to represent the unrepresented input vector.

3.3.4 Worth

The credit-assignment methods discussed so far deal with the assignment of errors to the units of a network. The errors indicate how the weights of a unit should be modified. Opposed to these error-correction techniques is the *generate-and-test* approach to the search for good units. The generation of new units at a particular stage of the learning process can be based on current hypotheses concerning which units, in terms of weight values, are useful for solving a task. To evaluate a unit a measure of usefulness, or *worth*, must be devised.

Samuel, 1959

Though not described as a connectionist network, Samuel's polynomial evaluation function for the game of checkers can be viewed as a linear output unit, and the terms of the polynomial considered as the output of the hidden units of a two-layer network. The coefficients of the polynomial are analogous to the weights connecting the output of the hidden units to the output unit. The coefficients of the polynomial, i.e., the weights of the output unit, are adjusted to reduce the difference between the evaluation of the current board configuration and the evaluation of other configurations encountered in a short look-ahead search.

A hidden unit is said to be of little worth in correcting the evaluation if it has developed an output weight of low magnitude, in which case it is discarded. New units are generated by selecting new features from a predefined list of features, chosen by Samuel as possibly being useful for the game of checkers. New units are initially assigned output weights of zero. The feature list includes combinational functions of two features.

Selfridge, 1959

For systems with more than one output unit, a way of collapsing a hidden unit's multiple measures of worth from each output unit must be used. A common measure is simply the sum of the hidden unit's output weights, as proposed by Selfridge for his Pandemonium system. The units of Pandemonium's hidden layer extract features with which the output units classify the input. During the learning phase for the output units, the parameters of the hidden units are held constant. Selfridge did not specify a particular method for training the output units, but suggested using hill-climbing search methods for finding the best weights for the output units.[3]

Once this process converges on the best set of weights or reaches some stopping criterion, modifications are made to the hidden units. First, units of low worth are discarded. New units are generated as described in the following section.

Uhr and Vossler, 1963

Uhr and Vossler worked with pattern classification tasks using a system whose output layer contained as many units as there were classes of input patterns. Uhr and Vossler's solution to

[3]Selfridge discussed the pitfalls of using a hill-climbing search, such as converging on local minima or maxima in the evaluation function. It was hoped, however, that the search through the parameter space of only one layer would be more efficient than searching through the parameter space of the entire system.

combining multiple measures of worth is to average all of the output weights for a hidden unit. The removal of a hidden unit that is very helpful for one output unit but useless for every other unit is a potential danger with this approach.

Klopf and Gose, 1969

Klopf and Gose compared three methods for measuring the worth of the hidden units of a two-layer connectionist system that had one output unit. One method was the magnitude of a unit's output weight, as in Selfridge's Pandemonium system. The product of the output weight and the unit's output value was the second measure, and the third was the absolute value of the cross correlation between the hidden unit's output and the system's desired output. Their results show that the product of the output weight and the output value produced better performance than the output weight alone, and both were better than the cross correlation measure. The former result was expected, since the true effect of a hidden unit on an output unit is determined both by the output weight and the output value of the hidden unit.

Ivanhenko, 1971

The process of learning in a connectionist system can be described as one of approximating a desired input-output mapping. This abstract view was taken by Ivanhenko as he developed his Group Method of Data Handling (GMDH) algorithm (Ivanhenko, 1971; see Duffy and Franklin, 1975, for modifications and applications), although Ivanhenko did not present his method in a connectionist framework. In connectionist terms, he showed how systems could be constructed to approximate complex mappings using units of limited complexity, such as units that compute a quadratic function of two input components. Complex mappings are realized by adding additional layers, since each layer adds two degrees of complexity to the network's mapping.

The GMDH algorithm assumes that a set of desired input-output pairs are available a priori, which are divided into a training set and a testing set. The algorithm proceeds by developing one layer at a time. For the first layer, the original input components are divided into disjoint pairs and a unit created for each that computes a quadratic polynomial of the pair of input components. A regression technique is used to determine the coefficients of the polynomials that best approximate the desired mapping, defined by the input-output pairs in the training set. The testing set is then used to calculate the mean-square error of each polynomial, or unit, and those with an error above a specific threshold are discarded. (This is related to Klopf and Gose's third measure of worth, the correlation between a unit's output and the desired output of the system.) Now the outputs of the remaining units become the input components to the next layer and the procedure is repeated to determine the polynomials and their coefficients for the units in the second layer. The procedure ends when a certain degree of accuracy is achieved.

Ivanhenko's algorithm is basically a search for sets of polynomials of minimal complexity that approximate a given function. The search begins with quadratic functions and the complexity is increased in small steps. A combinatorial explosion is avoided by evaluating the current set of polynomials at each stage and only keeping those receiving good evaluations as potentially useful terms in functions of higher complexity. Unpromising alternatives are removed at early stages, thus implementing a kind of *beam search* (Barr and Feigenbaum, 1981).

Holland, 1975, 1986

Holland has developed the "bucket-brigade" algorithm for assigning credit among the rules of a production system. Each rule maintains its own measure of worth, referred to by Holland as its "strength", and the bucket-brigade mechanism passes quantities of strength among the rules, based on their activation.

Briefly, the bucket-brigade algorithm consists of the following steps. Upon activation a rule decreases its strength by a certain amount (called a "tax"), which is distributed in equal fractions (in the simplest version) to rules that played a role in activating the first rule. If a rule causes an external action resulting in an evaluation, then this evaluation is added to the rule's strength.

Strengths are used in generating new rules by means of Holland's "genetic algorithm", described in the next section. Rules of low strength are discarded.

3.4 Modification

The manner in which a unit's weights are modified depends on the form of the credit. Three main categories of modification methods can be distinguished by the form of credit involved:

- a fraction of the gradient, whether exact or approximate;
- the minimal change indicated by the assigned credit;
- a change in the parameters of the generation procedure for new units.

Each category is reviewed below. Less emphasis is placed on distinguishing the authors, since methods of modifying weights are less varied than are the credit-assignment methods.

3.4.1 Fraction of Gradient

Gradient-descent procedures were presented in the previous section as methods for assigning credit. When the assigned credit is a sample of a true or approximate gradient, the adjustment to the weights is simply a fraction of the calculated gradient value. The fraction is typically a parameter, called ρ in the update equations given in the previous section, whose optimal value depends on the task and the network structure.

Methods for varying ρ have been considered as ways of speeding the convergence of the weights when ravines and other difficulties are presented by the gradient. Saridis (1970) has shown how values of ρ for each weight can be dynamically adapted to accelerate the convergence of gradient descent and stochastic approximation methods. A similar approach was studied by Sutton (Barto and Sutton, 1981b, Appendix C). The momentum term of Rumelhart, et al., discussed in the previous section, has an effect similar to that of changing ρ, although the differences have not been analyzed.

3.4.2 Minimal Change

Some systems that use the minimal change principle to assign errors adjust the weights in one step to remove the errors. Reilly, et al., and Soklic viewed this modification process as the shrinking or expanding of the units' domains to remove the current situation from or include it in the domains. The method of assigning errors used by Hampson is similar to what Reilly, et al., and Soklic used for their prototype units. However, Hampson differed in how modifications are made to the units. Rather than making an adjustment that was guaranteed to remove the error, he made relatively small adjustments in the weights, relying on repeated adjustments to guide the placement and size of the units' domains.

Stafford used a centrally-controlled mechanism to identify errors in the units' outputs. Once the errors are identified, the weights are adjusted by an amount necessary to remove the errors. Stafford's approach is unique in that many iterations of error-assignment and modification might be involved for the presentation of a single input.

Widrow assigned errors by looking for minimal changes, but, like Hampson, made relatively small changes in the weights. His modification method was the same as that used in his Adaline units:

$$\Delta w_{ij} = \rho \, (d_j - w^T x) \, x_i,$$

where d_j is the desired output assigned to hidden unit j as a member of a minimal set of hidden units to be changed.

3.4.3 Generate New Units

Some methods for learning in multilayer networks assume a fixed number of units and apply learning algorithms to every unit on every step. Others assume an initial set of units that are added to as experience is gained. This distinction is fuzzy—large modifications to a unit's weights can be considered as the generation of a new unit and the removal of an existing unit. In this section, it is convenient to focus on the generation of new units.

Methods for generating units can be classified as either *data-directed* or *model-directed*,[4] depending on whether the current input and outcome, or the current set of weight values are used to initialize the weights of the new unit.

Data-Directed Generation

Systems employing prototype units can encounter situations where the domains of the current set of units have been shifted or shrunk to the extent that no unit responds to the current input. A new unit must be generated if the system is to produce the desired output. Reilly, et al., initialize the new unit such that its domain is centered on the current input and extends for some radius. They decrease the initial radius as more units are added, reasoning that finer discriminations need to be made as experience accumulates.

Soklic also adds units whose domains are centered on the current input, but the shape (rectangular dimensions) of a new unit's initial domain is matched to the shape of nearby domains, based on the assumption that the neighboring shape is optimal for the part of the input space surrounding the current input.

In Uhr and Vossler's pattern classification system, a hidden unit's parameters define a binary mask which is scanned across an image, and a unit's output consists of the number of matches and the average location of the matches. They tested several methods for generating new units, one of which is data-directed. When the system's classification is in error, a mask is extracted from the image at a random position and generalized slightly by changing the parameters along the borders between regions of 0's and 1's to "don't-care" symbols, and replacing all remaining 0's and 1's with "don't-care" symbols with a probability of 1/2.

Model-Directed Generation

Uhr and Vossler (1963) and Klopf and Gose (1969) tested a simple process for generating units: units are formed by initializing their parameters to random values. This can be viewed as using a trivial model, devoid of knowledge about the task. These authors state that this method is not very efficient and that the generation process should be guided by past experience with a task.

Model-directed generation is intuitively an evolutionary process, with the premise that better units can be found by making small changes to current units of high worth. New units are tested and discarded if found to be of low worth. Selfridge (1959) followed this intuition in specifying the generation process for Pandemonium. First he discards units of low worth. These are replaced in two ways, called "mutated fission" and "conjugation". Mutated fission consists of randomly selecting one of the remaining hidden units and altering some of its weights, also at random. To form a new unit by conjugation, two of the remaining units are randomly chosen and logically combined by selecting one of ten non-trivial, logical functions and adding a unit that computes this logical function on the output of the two chosen units.

The conjugation method was also used by Uhr and Vossler (1963) and Ivanhenko (1971). Uhr and Vossler generated new units by randomly choosing two units and logically combining their input masks. Ivanhenko generated a unit for every pairwise product of the outputs of current units after discarding the units of low worth.

Holland has carried the evolutionary view even further with the development of his "genetic algorithm". The rules of his production system are represented as bit strings. To generate new rules a set of three operations are applied to rules of high worth, indicated by a rule's strength. The

[4]The labels *data-directed* and *model-directed* are adapted from similar uses in the machine learning literature (Cohen and Feigenbaum, 1981).

"mutation" operation is similar to Selfridge's "mutated fission": a high-worth rule is copied and its bits are flipped with a low probability. The "crossover" operation is performed by randomly choosing two rules of high worth, then randomly selecting a position in the rules' bit strings and interchanging the substrings following or preceding that position, resulting in four new rules. A third operation is called "inversion", which reorders the bits of a string.

The genetic algorithm generates rules based on the current hypotheses about which rules are useful, and is therefore a model-directed approach. Multiple rules are generated on each step and are tested through further experience with the task. The operation of the genetic algorithm has been extensively analyzed (Holland, 1975) and has been applied in several domains (Gillies, 1985; Goldberg, 1983; Smith, 1983).

3.5 Outline

We conclude this review with an outline of the multilayer learning methods described in this chapter. The direct search methods are outlined first, then the remainder are grouped according to their approach to the credit-assignment problem. The format of the outline is adapted from that used by Smith, Mitchell, Chestek, and Buchanan (1977) for their review of single and multilayer learning systems.

3.5.1 Direct Search

```
Direct Search

SYSTEM INPUT, OUTPUT: real-valued vector in, type of output irrelevant
STRUCTURE: multiple layers of units using parameterized, nonlinear functions;
         unrestricted architecture
CRITIC OUTPUT: scalar evaluation
TIMING OF LEARNING: after a number of interactions with task environment, every
         parameter is modified at once
ALL LAYERS
  GENERATION: new parameter vector is generated from:
         a) a uniform probability distribution (unguided random search)
         b) a Gaussian (or similar) probability distribution centered on
            the currently best parameter vector (guided random search)
         c) some function of the previously best n parameter vectors and
            their evaluations (deterministic search)
  CREDIT ASSIGNMENT: the critic assigns a scalar evaluation to each parameter
        vector
  MODIFICATION:  parameter vectors receiving high evaluations replace those of
        low evaluation
```

3.5.2 Gradient Methods

```
Rumelhart, Hinton, and Williams

SYSTEM INPUT, OUTPUT: real-valued vector in, real-valued vector out
```

STRUCTURE: multiple layers of units using real-valued, logistic function; unrestricted architecture

CRITIC OUTPUT: desired outputs for output units

TIMING OF LEARNING: information must flow from output layer backwards through hidden layers

OUTPUT LAYER:

  CREDIT ASSIGNMENT: every unit is assigned an error by the critic

  MODIFICATION: based on gradient

HIDDEN LAYERS:

  CREDIT ASSIGNMENT: gradient recursively calculated from a unit's output and the weights connecting its output to other units and those units' gradients

  MODIFICATION: based on gradient

Rosenblatt

SYSTEM INPUT, OUTPUT: real-valued vector in, binary components out

STRUCTURE: multiple layers of linear threshold units, acyclic

CRITIC OUTPUT: desired output for output units

TIMING OF LEARNING: information must flow from output layer backwards through hidden layers

OUTPUT LAYER:

  CREDIT ASSIGNMENT: every unit is assigned an error by the critic

  MODIFICATION: one of Rosenblatt's weight update rule

HIDDEN LAYER:

  CREDIT ASSIGNMENT: signed errors are assigned probabilistically based on error, hidden layer's output, and interconnecting weight; errors assigned with a low probability even when output layer's output is correct. A hidden unit's errors from every output unit are summed.

  MODIFICATION: one of Rosenblatt's weight update rule

Alder

SYSTEM INPUT, OUTPUT: real-valued vector in, binary components out
STRUCTURE: multiple layers of linear threshold units, acyclic
CRITIC OUTPUT: desired output for output units
TIMING OF LEARNING: only a single unit is modified at each step
ALL LAYERS:
  CREDIT ASSIGNMENT: a unit is picked at random
  MODIFICATION: Rosenblatt's perceptron weight update rule

Ackley, Hinton, Sejnowski

SYSTEM INPUT, OUTPUT: binary-valued vector in, binary-valued vector out
STRUCTURE: multiple layers of stochastic, linear threshold units; connections are
         symmetric; otherwise unrestricted architecture
CRITIC OUTPUT: desired output for visible units
TIMING OF LEARNING: simultaneously modified
ALL LAYERS:
  CREDIT ASSIGNMENT: gradient approximated by multi-step stochastic search in
         ``free'' and ``clamped'' modes
  MODIFICATION: based on approximate gradient

Farley and Clark

SYSTEM INPUT, OUTPUT: real-valued vector in, binary-valued vector out
STRUCTURE: multiple layers of stochastic, linear threshold units (with dynamics
         modeled after the neuron); unrestricted architecture
CRITIC OUTPUT: reinforcement
TIMING OF LEARNING: simultaneously
ALL LAYERS
  CREDIT ASSIGNMENT: gradient is approximated by single-step stochastic search
  MODIFICATION: based on approximate gradient

Barto, et al.

SYSTEM INPUT, OUTPUT: real-valued vector in, binary-valued vector out
STRUCTURE: multiple layers of stochastic, linear threshold units;
         otherwise unrestricted architecture
CRITIC OUTPUT: reinforcement
TIMING OF LEARNING: simultaneously
ALL LAYERS:
  CREDIT ASSIGNMENT: gradient is approximated by single-step stochastic search
  MODIFICATION: based on approximate gradient

3.5.3 Minimal Change

```
Widrow

SYSTEM INPUT, OUTPUT: real-valued vector in, binary components out
STRUCTURE: two layers of linear threshold units, strictly layered
CRITIC OUTPUT: desired output for output units
TIMING OF LEARNING: hidden layer modified first, then output layer if errors remain
HIDDEN LAYER:
  CREDIT ASSIGNMENT: all units are examined to find the set of units that can be
         minimally modified to reduce the output layer's errors
  MODIFICATION: Widrow's Adaline weight update rule
OUTPUT LAYER:
  CREDIT ASSIGNMENT: every unit is assigned an error by the critic
  MODIFICATION: Widrow's Adaline weight update rule

Stafford

SYSTEM INPUT, OUTPUT: real-valued vector in, binary components out
STRUCTURE: multiple layers of linear threshold units, single unit in output layer,
         all weights but those in very first layer restricted to positive values,
         acyclic architecture
CRITIC OUTPUT: desired output for output unit
TIMING OF LEARNING: determined by credit assignment procedure
ALL LAYERS:
  CREDIT ASSIGNMENT: all units are manipulated by systematically changing their
         output, assigning errors to units when their output and the output
         unit's output switch at the same time
  MODIFICATION: weights changed to values necessary for unit's output to switch
         back to original value

Soklic

SYSTEM INPUT, OUTPUT: real-valued vector in, binary components out
STRUCTURE: hidden layer is of prototype units having rectangular domains, output
         layer is of linear threshold units, weights have values of either 0 or 1
CRITIC OUTPUT: desired outputs for output units
TIMING OF LEARNING: hidden layer modified first, output layer only modified when new
         hidden unit is added
HIDDEN LAYER:
  CREDIT ASSIGNMENT: blame is assigned to units whose domains contain the situation
         and that are connected to the wrong output units, credit is assigned to
         units connected to correct output units and whose domains are close to
         the situation
  MODIFICATION: rectangular domains of bad units are reduced and shifted away from
         situation, good units are shifted towards situation, new unit is added with
```

domain centered on situation if no good units are close enough to situation
OUTPUT LAYER:
CREDIT ASSIGNMENT: every unit is assigned an error by the critic
MODIFICATION: when new hidden unit is added, weight of 1 is assigned to connection between new unit and correct output unit, 0's to other connections

Reilly, Cooper, and Elbaum

SYSTEM INPUT, OUTPUT: real-valued vector in, binary components out
STRUCTURE: hidden layer is of linear threshold units operated as prototype units having circular domains by normalizing input vectors; output layer is of linear threshold units, weights have values of either 0 or 1
CRITIC OUTPUT: desired outputs for output units
TIMING OF LEARNING: hidden layer modified first, output layer only modified when a new hidden unit is added
HIDDEN LAYER:
CREDIT ASSIGNMENT: blame is assigned to units whose domains contain the situation and that are connected to wrong output units, credit is assigned to units connected to correct output units and whose domains are close to the situation
MODIFICATION: circular domains of bad units are reduced, a new unit is added with domain centered on situation if no good units are close enough to situation
OUTPUT LAYER:
CREDIT ASSIGNMENT: every unit is assigned an error by the critic
MODIFICATION: when new hidden unit added, weight of 1 is assigned to connection between new unit and correct output unit, 0's to other connections

Hampson

SYSTEM INPUT, OUTPUT: vector of binary numbers in, vector of binary numbers out
STRUCTURE: hidden layer is of linear threshold units operated as prototype units by dealing only with binary input; output layer is of linear threshold units
CRITIC OUTPUT: desired outputs for output units
TIMING OF LEARNING: simultaneously
HIDDEN LAYER:
CREDIT ASSIGNMENT: blame is assigned to units whose domains contain the situation and that are connected to wrong output units
MODIFICATION: domains of bad units are reduced, units also competitively tune to each situation to ensure that all situations are contained in at least one unit's domain
OUTPUT LAYER:
CREDIT ASSIGNMENT: every unit is assigned an error by the critic
MODIFICATION: weights are strengthened based on probability of co-occurrence of corresponding input and desired action of value 1, weights are reduced by slowly decaying

3.5.4 Worth

Samuel

SYSTEM INPUT, OUTPUT: real-valued vector in, real-valued number out
STRUCTURE: hidden layer is units of nonlinear functions; output layer is a single
linear unit; strictly layered architecture
CRITIC OUTPUT: desired output of output unit
TIMING OF LEARNING: output unit is modified for set number of interactions with the
task environment, before hidden layer is modified
OUTPUT LAYER:
CREDIT ASSIGNMENT: correlations between signs of input components and sign of
the output unit's error
MODIFICATION: weight for input with largest correlation set to maximum value and
other weights set to proportional values, depending on the correlations
of their associated inputs
HIDDEN LAYER:
GENERATION: new units are taken from a predefined list of units that Samuel guessed
would be useful
CREDIT ASSIGNMENT: a unit is deemed as replaceable when its connection weight
to the last layer has been lower than all others for a set number of steps
MODIFICATION: replaceable units are placed at the bottom of the predefined list
of units

Selfridge

SYSTEM INPUT, OUTPUT: real-valued vector in, real-valued vector out
STRUCTURE: hidden layers are nonlinear, parameterized functions; output layer is of
linear units; strictly layered architecture
CRITIC OUTPUT: unspecified, must be able to derive a scalar evaluation from the
critic's output
TIMING OF LEARNING: last layer is modified until optimized for given parameters
of hidden units, before hidden units are modified
HIDDEN LAYERS:
GENERATION: new units are generated by:
a) randomly altering the parameters of an existing unit (``mutated fission''),
b) combining the functions of two existing units
CREDIT ASSIGNMENT: sum of weights connecting a unit to the output units
MODIFICATION: existing units are not modified, units having low-magnitude weights
connecting them to the last layer are removed

Uhr and Vossler

SYSTEM INPUT, OUTPUT: binary-valued vector in, binary-valued vector out
STRUCTURE: hidden layer is of units producing several values, based on scanning the
unit's binary-valued mask across the input image and looking for matches;

output layer is of prototype units, whose parameters were compared to the output of the hidden layer, with ‘‘amplifiers’’ associated with each component; strictly layered architecture

CRITIC OUTPUT: desired output for output units

TIMING OF LEARNING: output layer is modified whenever in error, hidden layer is modified when unit found to be of little use

OUTPUT LAYER:

CREDIT ASSIGNMENT: errors to each unit from critic

MODIFICATION: parameters updated based on their difference from output of hidden layer

HIDDEN LAYER:

GENERATION: new units are either prespecified or generated by:

a) by randomly assigning binary values to the mask,

b) randomly extracting a part of the input image, or

c) logically combining two existing units

CREDIT ASSIGNMENT: a unit becomes replaceable when its corresponding amplifier value, averaged over all output units, is low

MODIFICATION: replaceable units are removed

Klopf and Gose

SYSTEM INPUT, OUTPUT: real-valued vector in, real-valued vector out

STRUCTURE: hidden layer is of units using tabulated response functions, with one value for each different input vector; output layer is of single linear unit; strictly layered architecture

CRITIC OUTPUT: desired outputs for output units

TIMING OF LEARNING: output layer, then hidden layer

OUTPUT LAYER:

CREDIT ASSIGNMENT: errors to each unit from critic

MODIFICATION: optimal parameters directly calculated

HIDDEN LAYER:

GENERATION: new units are generated by randomly assigning output values to every input vector

CREDIT ASSIGNMENT: tried several ways:

a) magnitude of output weight,

b) product of a unit’s output and its output weight, and

c) absolute value of cross correlation between unit’s output and the desired output

MODIFICATION: units are not modified, number removed prespecified

Ivanhenko

SYSTEM INPUT, OUTPUT: real-valued vector in, real-valued number out

STRUCTURE: hidden layers are of units formed as products of two inputs; output layer is of a single linear unit; strictly layered architecture

CRITIC OUTPUT: desired outputs for output units

TIMING OF LEARNING: first hidden layer's units generated, output unit is optimized, second hidden layer is generated, output layer is optimized, etc.

OUTPUT LAYER:

CREDIT ASSIGNMENT: errors to the unit from critic

MODIFICATION: optimal parameters directly calculated

HIDDEN LAYERS:

GENERATION: new units are generated by forming products of every possible pairing of components generated from the previous layer

CREDIT ASSIGNMENT: magnitude of weight connecting unit to output unit

MODIFICATION: units are not modified, units removed if result in small weight connection to output unit

Holland

SYSTEM INPUT, OUTPUT: binary-valued vector in, binary-valued vector out

STRUCTURE: set of production rules, conditions are strings of 0, 1, and ''don't-care'' symbols, actions are strings of 0 and 1; each rule has a numerical strength

CRITIC OUTPUT: scalar evaluation

TIMING OF LEARNING: rule base modified after set number of interactions with task environment

ALL LAYERS:

GENERATION: new rules formed by genetic-algorithm (randomly choosing rules based on their strengths and applying genetic operators to them, producing new rules)

CREDIT ASSIGNMENT: a rule's strength is altered when it activates another rule through the bucket-brigade algorithm

MODIFICATION: rules of low strength are removed

Chapter 4

Comparison of Methods for Learning Missing Features

Experiments with the multiplexer task in Chapter II focused on the issue of developing new features that are required for a task's solution. It was shown that with the original representation the task could not be solved by a single-layer network of linear threshold units. In this chapter, eleven algorithms for learning new features through modifications to the weights of hidden units are evaluated by how well they overcome the difficulties presented by the original representation.

The multiplexer task was defined in Chapter II, so the description here is brief. The input vectors consist of two address bits and four data bits, plus a constant component of 0.5. The desired output is given by a multiplexer function of the address and data bits. If we call the address components a_1 and a_2 and the data components d_1, d_2, d_3, and d_4, the expression for the multiplexer function is

$$\bar{a}_1\bar{a}_2d_1 \vee \bar{a}_1a_2d_2 \vee a_1\bar{a}_2d_3 \vee a_1a_2\ d_4.$$

There are a total of 2^6, or 64, input vectors.

The system used in the following experiments is an extension of the single-layer system applied to this task in Chapter II. An additional layer was formed by adding four units, each receiving input from the environment and generating a new input component to the output layer. This structure is shown in Figure 4.1. By supplying the original input components to the output unit, linear associations can be formed directly from the input to the network's output, permitting some of the input-output associations to be learned without the adjustment of hidden-unit weights. Note that the system contains no cycles. The outputs of Units 1 through 4 are always calculated before the output of Unit 5.

For the experiments to be described, the hidden-unit algorithm is the primary variable. The learning algorithm for the output unit was the same for most experiments. The perceptron algorithm (Rosenblatt, 1962) was used for the output unit since it is well-known and was found in Chapter II to be relatively insensitive to the parameter ρ (so ρ would not have to be varied to optimize performance). The application of Rumelhart, et al.'s error back-propagation algorithm (Rumelhart, Hinton, and Williams, 1986) to the hidden units requires the use of a differentiable output function in the output unit, so Rumelhart, et al.'s semilinear output function and learning algorithm, described in Chapter III, were used in the output unit for the experiments with the error back-propagation algorithm.

A step in the simulation of this system consists of the following. An input vector is selected by choosing one randomly, without replacement, from the set of all input vectors. Upon receipt of an input vector, the outputs of the hidden units are calculated, followed by the calculation of the output of the output unit. The output of the system, equal to that of the output unit, is subtracted from the desired output. This error controls the perceptron learning algorithm as it is applied to the weights of the output unit, after which the particular learning algorithm being tested in the hidden units is applied to the hidden units' weights (though some algorithms, such

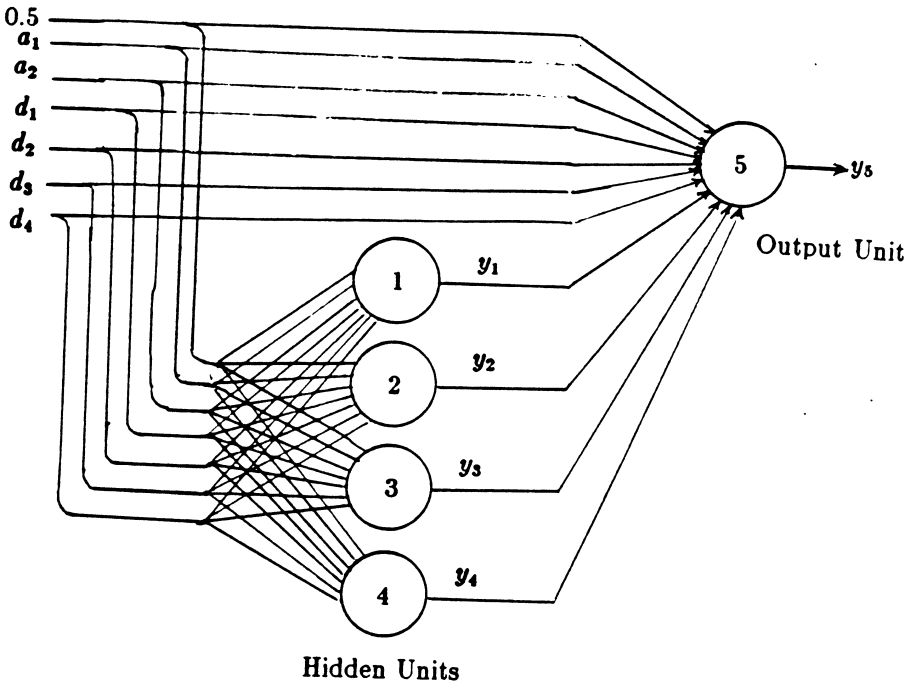


Figure 4.1: Two-Layer Network for Multiplexer Experiments

as the direct-search algorithms, do not change the weights of the hidden units on every step). This completes one step in the simulation. Every input vector is presented once during the first 64 steps, and once again for every subsequent set of 64 steps, where the order of presentation is determined randomly.

We now present the algorithms along with the experimental results. The direct-search algorithms are presented first. These algorithms require no knowledge about the system other than the number of hidden-unit weights and their range of values. Following the direct-search algorithms, several error back-propagation algorithms are presented that involve the propagation of the output unit's error to the hidden units. Then some reinforcement-learning algorithms are presented, which effectively give each hidden unit the responsibility of conducting a search of its own weight space. A modification of one reinforcement-learning algorithm is considered that generates localized reinforcements to the hidden units by propagating information from the output unit back to the hidden units. Finally, a mechanism is added that treats hidden units that have not yet acquired a substantial influence on the output unit differently from those that do have influence, to reduce the time needed to discover useful new features.

The behavior of each algorithm depends on several parameters. A comparative study should guarantee that parameter values are used that are optimal for a given algorithm to ensure the absence of bias in favor of one algorithm over another. However, the time required to simulate the connectionist system used in these experiments prohibited a thorough optimization of the parameter values. We were only able to test an average of six different values over a broad range for each parameter, and when an algorithm depended on more than one parameter only one parameter was varied at a time.

4.1 Direct-Search Algorithms

4.1.1 Unguided Random Search

The first algorithm considered is the simplest possible random search. It consists of randomly choosing new values from a uniform probability density function for the weights of every hidden unit in the system. The set of values is evaluated by letting the system interact with the environment for a number of steps, remembering the set of values receiving the best evaluation. When considering such a search as an optimization method, one usually assumes the state of the parameterized system remains constant during its evaluation. Here we want to evaluate the current values of the weights by measuring the degree to which the system can solve the task using the given weight values, so the output unit continues to learn while the weights of the hidden units are held constant. The weights of the output unit are set to zero whenever new values for the hidden units' weights are generated.

To describe this in more detail we must define some terms. Let w_h be a vector of weight values for the hidden units, w_o be a weight vector for the output unit, $y_j[t]$ be the output of Unit j, and let $d_j[t]$ be the desired output for Output Unit j (or the desired value of $y_5[t]$ in this case), where t is the current time step. There are four hidden units, each receiving seven input components, so w_h contains 28 components. The single output unit receives seven input components plus four from the output of the hidden units, giving w_o 11 components. Let n be the number of steps over which each vector of weight values is evaluated, and let w_h^* and e^* be the currently-best weight vector and its evaluation, and w_o^* be the vector of weights of the output unit that developed during the evaluation of w_h^*. With this notation we can now present the algorithm:

Unguided Random Search Algorithm

1. Generate a new weight vector to be used during the next n steps:

$$w_h = \text{random variable from uniform probability density function over } [-1, 1]^{28},$$

$$w_o = 0.$$

2. Interact with the environment for n steps, producing the sequences $y_i[k]$, and $d_j[k]$, as $k = t, t+n$, for all units i and all output units $j \in O$, allowing the output units to learn after each step. Recall that O is the set of indices of the output units, so $O = \{5\}$ for the multiplexer task. All units use the linear threshold output function.

3. Evaluate the weight vector:

$$e = \sum_{k=t}^{t+n} \sum_{j \in O} |d_j[k] - y_j[k]|,$$
$$t = t + n.$$

   If this is the first evaluation, then let $w_h^* = w_h$, $w_o^* = w_o$, and $e^* = e$ and repeat, starting with Step 1. Otherwise, continue.

4. If the number of errors is lower than the previously-best evaluation, then the current weight vector becomes the best vector:

$$\begin{array}{lll} \text{if} & e < e^* & \\ & \text{then} & w_h^* = w_h \\ & & w_o^* = w_o \\ & & e^* = e. \end{array}$$

5. Repeat, starting with Step 1, for a prespecified number of steps.

6. At the conclusion of the prespecified number of steps, assign the "best" weight vectors to the hidden and output units, before calculating the final-step performance measure ν.[1]

The unguided random search was tested on the multiplexer task for several values of n. For each value of n, the results from 10 runs of 300,000 steps each were collected. The final performance level of a run, ν, is calculated by first substituting, for the final-step weight vector, the weight vector determined as the best for that run. Then the number of possible input vectors for which the network generated an incorrect output is tallied. As before, ν can range from 0 to 64, and a purely random strategy of generating outputs would result in an average value of 32 for ν.

In addition to the performance level at the final step of each run, we determined the value of the cumulative performance, μ, which for a single run is the sum of the number of errors made on every step. For a nonlearning, random strategy, errors would occur on an average of half of the steps, producing a value for μ of 150,000.

The results of the experiments are listed in Table 4.1, including the 99% confidence intervals of ν and μ. The unguided random search performed better than a nonlearning, random strategy for all values of n that were tried. The value of μ consistently declines as the parameter n increases. Recall that after every n steps, a new weight vector is generated that does not depend on previously-tested vectors, so there is no gradual improvement in performance as a run progresses. However, since the output unit is learning throughout each n step period, larger values of n result in better performance at the end of the n step period and better average performance over that period, which explains the inverse relationship of μ and n.

The values of ν do not indicate any one value of n as being optimal. When n is 200 or less, significantly higher values of ν are obtained than when n is 400 or greater. In fact, for $n \leq 200$, performance is not significantly different from that of a single layer, for which $\nu \approx 24$.

A learning curve for the unguided random search on the multiplexer task was obtained by choosing the best value of n, which is 1600, and performing 30 runs of 300,000 steps each. This resulted in performance measures of $\nu = 17.0 \pm 2.93$ and $\mu = 115,062 \pm 229$ and the learning curve in Figure 4.2 (appearing again in Figure 4.5 on page 86). Figure 4.2 includes learning curves for

[1] The performance measures μ and ν are defined in Chapter II.

n	ν	μ
50	25.6 ± 2.78	$140,228 \pm 263$
100	22.7 ± 2.88	$134,397 \pm 128$
200	23.4 ± 3.80	$127,913 \pm 237$
400	18.0 ± 3.21	$122,209 \pm 176$
800	16.5 ± 2.66	$117,899 \pm 342$
1600	15.7 ± 3.22	$115,099 \pm 324$
3200	18.4 ± 5.23	$112,848 \pm 462$
6400	17.0 ± 4.29	$112,577 \pm 803$
12800	16.9 ± 3.96	$111,477 \pm 1,064$
25600	17.7 ± 3.71	$110,654 \pm 1,535$

Table 4.1: Unguided Random Search on the Multiplexer Task

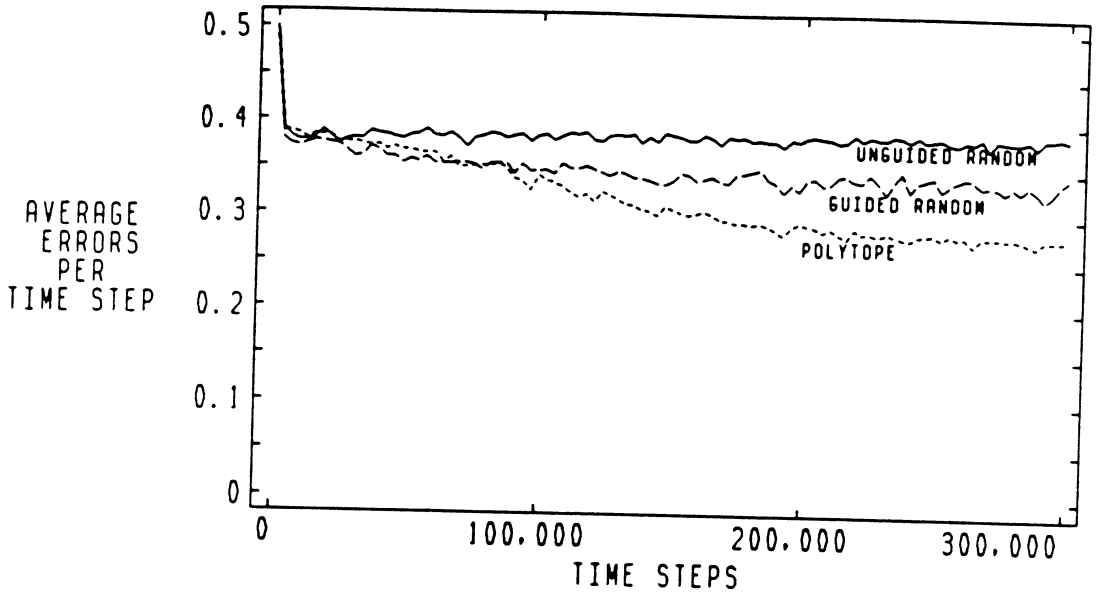


Figure 4.2: Learning Curves of Direct-Search Algorithms on the Multiplexer Task

the other direct search algorithms described below. On this and all subsequent graphs, an initial quick drop appears from 0.5 errors per step to approximately 0.37 or 0.38. This is caused by the output unit learning as many correct responses as possible; a single unit given the input vectors for the multiplexer task can learn the correct output for 40 of the 64 input vectors, resulting in an average of 0.375 errors per step.

4.1.2 Guided Random Search

When an improvement in performance is desired during the learning process, as in cases where a predefined finite training set of input vectors is not available, the unguided random search cannot be used. The generation of weight vectors must be related to the currently-best vector in a way that results in a level of performance that is better, on average, than that achieved for the previously-tried vector.

There are many ways of making the generation of weight vectors dependent on the currently-best vector (or on a series of best vectors). We studied two methods, the first being a guided random search and the second the polytope method described in the next section. The guided ran-

n	ν	μ
50	27.2 ± 3.76	$138,978 \pm 898$
100	24.1 ± 3.61	$131,957 \pm 2,102$
200	18.4 ± 3.94	$124,089 \pm 1,345$
400	13.8 ± 3.76	$115,390 \pm 3,271$
800	13.3 ± 4.73	$111,205 \pm 3,851$
1600	13.1 ± 4.21	$106,544 \pm 3,314$
3200	12.5 ± 3.94	$106,818 \pm 2,524$
6400	16.5 ± 4.48	$108,225 \pm 2,413$
12800	17.6 ± 5.29	$108,620 \pm 3,615$

$\tau = 2$

τ	ν	μ
0.1	18.9 ± 4.91	$106,894 \pm 6,204$
0.2	17.1 ± 2.53	$109,454 \pm 4,897$
0.5	14.9 ± 4.13	$105,343 \pm 6,124$
1.0	11.4 ± 3.58	$102,583 \pm 6,128$
2.0	12.5 ± 3.94	$106,818 \pm 2,524$
4.0	15.6 ± 2.83	$108,128 \pm 2,797$
8.0	15.0 ± 4.06	$108,498 \pm 3,066$

$n = 3200$

Table 4.2: Guided Random Search on the Multiplexer Task

dom search differs from the unguided random search only in the manner of generating new weight vectors. Rather than generating weight vectors from a uniform probability density function, they are chosen from a probability density function (defined below) centered on the currently-best weight vector. This density function is symmetric about the currently-best vector and the probability of selecting vectors decreases as the Euclidean distance from the currently-best vector increases. The algorithm is defined as follows.

Guided Random Search Algorithm

1. Generate a new weight vector to be used during the next n steps:

$$w_h = \text{random variable from probability density function } \Psi(w), \text{ given by: } \Psi(w) = \frac{1}{1 + e^{\frac{|w - w_h^*|}{\tau}}}$$

with mean w_h^* and "spread" τ (see below),

$$w_o = 0.$$

Steps 2 through 5 are identical to those listed for the Unguided Random Search Algorithm. (See page 65.)

This algorithm depends on two parameters: the number of steps between the generation of weight vectors is given by n, and the "spread" of the logistic probability density function is given by τ. The logistic function has been used to model the noise in distributed, dynamical systems and the parameter τ is referred to as the "temperature" of the system (see Chapter III, page 46).

As stated earlier, the amount of computer time required to perform these experiments prevented a systematic search for the optimal values of n and τ. However, we did perform two unidimensional searches by holding τ constant with a value of 2 while varying n, then varying τ while holding n constant with the value resulting in the best performance. For each parameter setting, results were averaged over 10 runs with each run lasting 300,000 steps.

The results in Table 4.2 show that the value of n must be neither too low nor too high to achieve good performance at the end of each run. However, unlike the results from the unguided random search, the cumulative performance measure μ also has a U-shape as n increases, showing evidence of a tradeoff between decreasing the error rate through learning in the output unit (large n) and through optimizing the weights of the hidden units by making more trials (small n).

Performance as a function of τ also has a U-shape; there appears to be an optimal value of τ in the range of 0.5 to 2. Note that as the value of τ increases, the logistic probability density

function comes to approximate a uniform density function; the guided random search will behave as the unguided random search for large values of τ. Performance being worse for larger values of τ indicates that the multiplexer task presents the learning system with situations for which following the gradient of the error function with respect to the system's weights is advantageous. The poor performance for low values of τ might be due to either:

- making weight changes whose magnitudes are smaller than those resulting from higher values of τ, thus not converging upon a good set of weights as quickly, or
- becoming "stuck" at locally optimum weight vectors, whereas the higher values of τ produce a higher probability of jumping out of locally optimum regions of the weight space.

The optimal value of τ would provide the right balance between these two effects.

This algorithm's learning curve in Figure 4.2 is averaged over 30 runs of 300,000 steps each, using $n = 3200$ and $\tau = 1$. The resulting performance levels are $\nu = 13.1 \pm 2.36$ and $\mu = 103,866 \pm 3420$.

4.1.3 Polytope Algorithm

Another method for directly searching the weight space is the Polytope Algorithm (Gill, Murray, and Wright, 1981). This method is often called the "simplex" method, not to be confused with the simplex method for linear programming. The polytope algorithm is a deterministic hillclimbing method that maintains a list of m weight vectors, ordered according to their evaluations. The m weight vectors are treated as vertices of a polytope in $m-1$–dimensional space, and new vectors are generated in a fashion designed to shift the polytope towards an optimum weight vector, taking large steps when progress is being made in improving the evaluation and taking smaller steps when it appears that the optimum has been approached. Since this is a deterministic hill-climbing method it can become stuck at locally optimum points. We included it in our study as an example of a reasonably sophisticated, deterministic, direct-search algorithm to complement the random methods presented above.

Our application of the polytope algorithm to learning in the hidden units is implemented as follows:

Polytope Algorithm

1. Generate m 28-dimensional weight vectors, w_i, randomly:

$$w_i = \begin{array}{l}\text{random variable from uniform probability} \\ \text{density function over } [-1,1]^{28}, \text{ for } i = 1, \ldots, m.\end{array}$$

   These define the vertices of a polytope in $m-1$–dimensional space.

2. Evaluate each weight vector w_i by fixing the hidden units' weights to w_i, setting the weights of the output units to zero, and interacting with the environment while learning in the output units, for $t = (i-1)n, \ldots, in$. All units use the linear threshold output function. Afterwards, the time step counter, t, is incremented:

$$\begin{aligned} w_h &= w_i, \\ w_o &= 0, \\ e_i &= \sum_{k=(i-1)n+1}^{in} \sum_{j \in O} |d_j[k] - y_j[k]|, \\ t &= in + 1. \end{aligned}$$

   In practice, t is advanced by 1 after every system-environment interaction.

3. Reorder e_i and the corresponding $w_i, i = 1, \ldots, n$ such that:

$$e_1 \leq e_2 \leq \cdots \leq e_m.$$

4. (Reflection Step) Find the centroid, c, of the $m-1$ best weight vectors, *reflect* the worst weight vector through the centroid to generate w_r, set the output units' weights to zero, and evaluate w_r, giving e_r:

$$\begin{aligned} c &= \frac{1}{m-1} \sum_{i=1}^{m-1} w_i, \\ w_r &= c + \rho_r (c - w_m), \\ w_h &= w_r, \\ w_o &= 0, \\ e_r &= \sum_{k=t}^{t+n} \sum_{j \in O} |d_j[k] - y_j[k]|, \\ t &= t + n. \end{aligned}$$

Replace the worst weight vector, w_m, with the reflected vector, w_r, if w_r is not better than w_1 and not worse than w_{m-1}:

$$\text{If } e_1 \leq e_r \leq e_{m-1}$$

$$\begin{aligned} \text{then } w_m &= w_r \\ e_m &= e_r. \end{aligned}$$

If w_m was replaced, repeat, starting with Step 4; otherwise continue to Step 5.

5. (Expansion Step) If the reflected weight vector is the new best vector, then the direction of reflection was very successful and an attempt is made to *expand* the polytope by generating w_e and evaluating it:

$$\text{if } e_r > e_1, \text{ then skip to Step 6.}$$

$$\begin{aligned} w_e &= c + \rho_e (w_r - c), \\ w_h &= w_e, \\ w_o &= 0, \\ e_e &= \sum_{k=t}^{t+n} \sum_{j \in O} |d_j[k] - y_j[k]|, \\ t &= t + n. \end{aligned}$$

Replace the worst vector with the better of the reflected and the expanded vectors:

$$w_m = \begin{cases} w_e, & \text{if } e_e < e_r; \\ w_r, & \text{otherwise,} \end{cases}$$

$$e_m = \min(e_e, e_r).$$

Repeat, starting with Step 4.

6. (Contraction Step) This step is performed when the reflected vector is either the worst ($e_r \geq e_m$) or the second to the worst vector ($e_r > e_{m-1}$). The polytope is *contracted* by

generating w_c in one of two ways:

$$w_c = \begin{cases} w_1 + \rho_c(w_m - w_1), & \text{if } e_r \geq e_m; \\ w_1 + \rho_c(w_r - w_1), & \text{if } e_m > e_r > e_{m-1}, \end{cases}$$

$$\begin{aligned} w_h &= w_c, \\ w_o &= 0, \\ e_c &= \sum_{k=t}^{t+n} \sum_{j \in O} |d_j[k] - y_j[k]|, \\ t &= t + n. \end{aligned}$$

If the contracted vector is better than both the reflected vector and the worst vector, then it replaces the worst vector:

$$\text{If } e_c < e_r \text{ and } e_c < e_m$$

$$\begin{aligned} \text{then} \quad w_m &= w_c \\ e_m &= e_c. \end{aligned}$$

Otherwise, the polytope is shrunk in the direction of the best vector and the resulting weight vectors are evaluated and reordered:

$$\begin{aligned} &\text{If } e_c > e_r \text{ or } e_c > e_m \\ &\quad \text{then} \quad \text{for} \quad i = 2, \ldots, m \\ &\qquad w_i = \frac{(w_1 + w_i)}{2}, \\ &\qquad w_h = w_i, \\ &\qquad w_o = 0, \\ &\qquad e_i = \sum_{k=t}^{t+n} \sum_{j \in O} |d_j[k] - y_j[k]|, \\ &\qquad t = t + n, \end{aligned}$$

$$\text{Reorder } w_i, \ i = 1, \ldots, m.$$

Repeat, starting with Step 4.

7. After the prespecified number of time steps have elapsed, the best weight vector is assigned to the weights of the system before calculating the final-step evaluation.

The polytope algorithm depends on the parameter m, the number of weight vectors maintained as vertices of the polytope, and the parameter n, the number of steps over which each weight vector is evaluated. Other parameters are ρ_r, ρ_e, and ρ_c, which determine the lengths of the reflection, expansion, and contraction steps, respectively. Valid ranges for these parameters are $\rho_r > 0$, $\rho_e > 1$, and $0 < \rho_c < 1$. To reduce the number of experiments to a practical level, we did not attempt to find optimal values for ρ_r, ρ_e, and ρ_c, but set $\rho_r = 2$, $\rho_e = 2$, and $\rho_c = 0.2$. We did vary m and n, as shown in Table 4.3. The value of m was fixed at 20 while n varied, after which n was fixed at 1600, which gave the best value of ν, while m was varied. The results are again averages over 10 runs at 300,000 steps per run.

Table 4.3 suggests that the optimum value of n is between 400 and 3,200. The results are even less conclusive about the optimum value of m; additional runs must be made to obtain performance averages with less variance. The values $n = 1600$ and $m = 10$ were used in 30 runs of 300,000 steps to obtain the learning curve in Figure 4.2, resulting in $\nu = 14.2 \pm 2.09$ and $\mu = 94,977 \pm 3079$.

n	ν	μ
200	20.8 ± 4.04	$118,780 \pm 6,537$
400	17.8 ± 4.33	$105,624 \pm 4,442$
800	13.0 ± 3.82	$99,575 \pm 5,319$
1,600	12.6 ± 2.70	$102,449 \pm 3,654$
3,200	14.2 ± 2.76	$109,711 \pm 1,460$
6,400	15.7 ± 3.74	$110,860 \pm 2,058$
12,800	19.0 ± 3.93	$110,866 \pm 2,488$

$m = 20$

m	ν	μ
3	17.4 ± 2.78	$100,046 \pm 6,767$
5	17.6 ± 4.51	$96,223 \pm 4,441$
10	12.1 ± 1.97	$94,157 \pm 4,165$
15	15.9 ± 6.15	$102,793 \pm 4,071$
20	12.6 ± 2.70	$102,449 \pm 3,654$
25	14.7 ± 4.24	$107,972 \pm 2,447$

$n = 1600$

Table 4.3: Polytope Algorithm on the Multiplexer Task

None of the direct-search methods were able to solve the multiplexer task within the allotted 300,000 steps. The unguided random search shows no improvement over time, because the weight vectors being tested are not dependent on previous search steps. Its final performance level is slightly better than that of the single-layer system ($\nu = 17$ versus $\nu = 24$). The guided random search does show improvement over time, though the slope of its learning curve becomes approximately flat early in the runs. Averaged over the last 3,000 steps of every run, the number of errors per step is approximately 0.35. The polytope algorithm performs better than both random search methods, reaching an average over the last 3,000 steps of 0.28 errors per step.

4.2 Error Back-Propagation Algorithms

Next we discuss some error back-propagation algorithms for learning in hidden units, starting with the algorithm due to Rosenblatt described in Chapter III.

4.2.1 Rosenblatt

Rosenblatt is well-known for his work with the perceptron-family of learning algorithms (Rosenblatt, 1962), but his error back-propagation algorithm has received little attention. Since this was proposed early in the history of work on learning in multilayer systems and seemed to work reasonably well for the experiments Rosenblatt performed, we wished to include it in our study. Rosenblatt's algorithm is a nondeterministic way to assign errors to hidden units based on the errors of output units. The following is our specification of Rosenblatt's algorithm:

Rosenblatt's Back-propagation Algorithm

1. Initialize all weights to zero:
$$w_h = 0,$$
$$w_o = 0.$$

2. Receive input vector, calculate the output of all units using a linear threshold function, and receive error signals for the output units.

3. Apply the perceptron learning algorithm to the output units.

4. Calculate the error, δ_{jk}, passed back from output unit k to hidden unit j (probabilistically based on the output unit's error, the weight connecting unit j to unit k, and the output of

ρ	ν	μ
0.030	24.7 ± 3.54	$121,085 \pm 238$
0.060	23.6 ± 2.96	$121,286 \pm 168$
0.125	22.8 ± 3.11	$121,112 \pm 156$
0.250	22.0 ± 2.92	$121,129 \pm 176$
0.500	21.9 ± 2.83	$121,175 \pm 278$
1.000	24.0 ± 2.74	$121,161 \pm 169$
2.000	23.1 ± 3.18	$121,132 \pm 134$

Table 4.4: Rosenblatt's Algorithm on the Multiplexer Task

unit j).

$$v_k[t] \;=\; \text{random variable from a uniform probability density function over } [0,1], \text{ where } k \in O \text{ takes the values of the indices of the output units,}$$

$$\delta_{jk}[t] = \begin{cases} -1, & \text{if } y_j[t] = 1 \text{ and } (d_j[t] - y_j[t])\, w_{jk}[t] < 0 \text{ and } v_k[t] < p_1; \\ +1, & \text{if } y_j[t] = 0 \text{ and } (d_j[t] - y_j[t])\, w_{jk}[t] > 0 \text{ and } v_k[t] < p_2 \\ & \qquad \text{or} \\ & \text{if } y_j[t] = 0 \text{ and } (d_j[t] - y_j[t])\, w_{jk}[t] \leq 0 \text{ and } v_k[t] < p_3; \\ 0, & \text{otherwise.} \end{cases}$$

5. Apply the perceptron learning rule to each hidden unit j, using the sign of the sum of the back-propagated errors from the output units as the error signals:

$$\Delta w_{ij}[t] \;=\; \rho \,\mathrm{sgn}\left(\sum_{k \in O} \delta_{jk}[t]\right) x_i[t].$$

6. Repeat, starting at Step 2, until the prespecified number of time steps has elapsed.

Rosenblatt's algorithm depends on the parameter ρ, a factor determining the magnitude of change for each weight, and the parameters p_1, p_2, and p_3, which are probabilities affecting the frequencies with which the back-propagated error signals take the values $+1$ and -1. Rosenblatt performed a number of experiments and determined that the values $p_1 = 0.9$, $p_2 = 0.3$, and $p_3 = 0.1$ were reasonable values. Rather than attempting to optimize all four parameters, we used these values for p_1, p_2, and p_3 for all experiments, only varying the value of ρ.

The results are in Table 4.4. There are few significant differences for different values of ρ, though values from 0.125 to 0.5 resulted in slightly lower values of ν. The values of ν and μ show no improvement over a single-layer system. Indeed, the learning curve for Rosenblatt's algorithm, in Figure 4.3, shows no improvement over time and is always worse than the single-layer level. The learning curve is averaged over 30 runs of 300,000 steps each, giving values of $\nu = 23.9 \pm 1.58$ and $\mu = 121,115 \pm 92$. To fairly judge the performance of Rosenblatt's algorithm, additional values of p_1, p_2, and p_3 must be tested.

4.2.2 Rumelhart, Hinton, and Williams

Another approach to the back-propagation of errors was taken by Rumelhart, Hinton, and Williams (1986). They used semilinear units, defined in Chapter III, for which the gradient of an error function with respect to each weight can be derived. Rumelhart, et al.'s algorithm is based on a recursive formulation of this gradient which is realized as a scheme of back-propagating errors. Our specialization of this algorithm to our two-layer system is as follows:

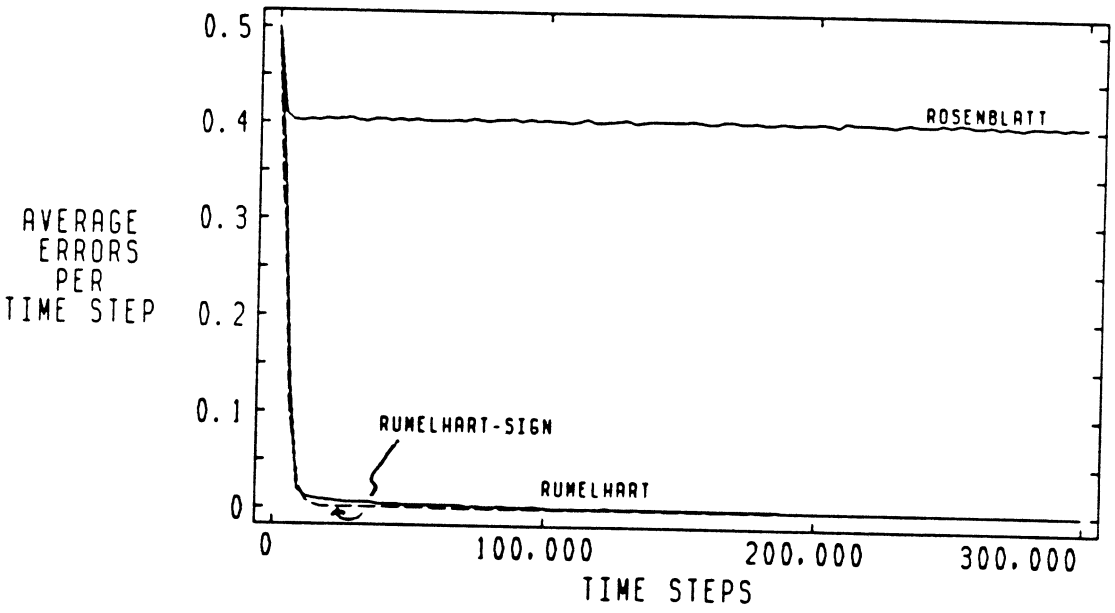


Figure 4.3: Learning Curve for Error-Correction Algorithms on the Multiplexer Task

Rumelhart's Back-Propagation Algorithm

1. Randomly initialize all weights:

$$w_h = \text{random variable from uniform probability density function over } [-0.1, 0.1]^{28},$$

$$w_o = \text{random variable from uniform probability density function over } [-0.1, 0.1]^{11}.$$

2. Receive input vector, calculate output of all units, and receive error signals for the output units. All units use the semilinear output function:

$$y_j[t] = \frac{1}{1 + e^{-\sum_{i=0}^{6} w_{ij}[t]\, x_i[t]}}.$$

3. Calculate δ_k for each output unit $k \in O$:

$$\delta_k[t] = (d'_k[t] - y_k[t])\ y_k[t]\ (1 - y_k[t])\,,$$

where d'_k is a modified version of the desired output, defined as

$$d'_k[t] = \begin{cases} 0.9, & \text{if } d_k[t] = 1; \\ 0.1, & \text{if } d_k[t] = 0. \end{cases}$$

ρ	ν	μ
0.05	35188 ± 63	19.8 ± 0.84
0.10	31716 ± 1602	11.7 ± 3.30
0.25	14144 ± 1426	0.3 ± 0.55
0.50	6966 ± 1052	0.3 ± 0.39
1.00	4944 ± 1224	0.7 ± 0.39
2.00	3289 ± 935	0.2 ± 0.52
4.00	3294 ± 836	0.2 ± 0.52
8.00	13446 ± 4097	6.6 ± 2.93
16.00	32422 ± 5497	18.3 ± 3.12

$\rho_m = 0$

ρ	ν	μ
0.05	34976 ± 496	18.9 ± 1.97
0.10	33218 ± 1671	15.9 ± 4.57
0.25	26245 ± 7354	9.3 ± 7.79
0.50	11287 ± 2562	0.1 ± 0.26
1.00	3836 ± 869	0.2 ± 0.52
2.00	3267 ± 1229	1.0 ± 0.86
4.00	8905 ± 2213	3.5 ± 1.55

$\rho_m = 0.5$

ρ	ν	μ
0.05	6130 ± 349	0.1 ± 0.26
0.10	3207 ± 454	0.0 ± 0.00
0.25	1747 ± 480	0.0 ± 0.00
0.50	1492 ± 844	0.2 ± 0.52
1.00	5802 ± 2686	1.9 ± 1.86

$\rho_m = 0.9$

Table 4.5: Rumelhart, et al.'s Algorithm on the Multiplexer Task

4. Apply the learning rule to the weights of each output unit k:

$$\Delta w_{jk}[t] = \rho\, \delta_k[t]\, x_j[t] + \rho_m\, \Delta w_{jk}[t-1],$$

where $x_j[t]$ is an input component received by output unit k. Recall that the output units receive the original input terms to the system plus the output of the hidden units.

5. Calculate δ_j for each hidden unit j:

$$\delta_j[t] = \left(\sum_{k \in O} \delta_k[t]\, w_{jk}[t] \right)\, y_j[t]\, \left(1 - y_j[t]\right).$$

6. Apply the learning rule to the weights of each hidden unit j:

$$\Delta w_{ij}[t] = \rho\, \delta_j[t]\, x_i[t] + \rho_m\, \Delta w_{ij}[t-1].$$

7. Repeat, starting with Step 2, until the prespecified number of time steps have elapsed.

By adding a fraction of the previous step's weight change, it is hoped that the weight values will be more likely to follow the slope of the error function at the bottom of steep valleys, by canceling opposing steps up one side or the other. Rumelhart, et al., consider this additional term as affecting the "momentum" of the trajectory of weight values, as described in Chapter III. The algorithm has two parameters, the rate of change parameter ρ and the factor ρ_m that controls the magnitude of the momentum term. Table 4.5 shows the values of ρ and ρ_m that were tested and the results averaged over 10 runs of 100,000 steps each.

Note the modification of the desired output value in Step 3. Rather than values of 1 and 0, values of 0.9 and 0.1 are used. Without this modification, weight values can grow in magnitude to the point where truncation errors due to the particular computer implementation can cause weight values to become frozen—the value of $y(1-y)$ in the weight update equation becomes equal to zero.

ρ	ν	μ
0.10	27759 ± 3438	7.2 ± 3.74
0.25	11594 ± 958	0.1 ± 0.26
0.50	5846 ± 1370	0.0 ± 0.00
1.00	3013 ± 573	0.1 ± 0.26
2.00	2336 ± 355	0.1 ± 0.26
4.00	4378 ± 1179	1.0 ± 0.86

$\rho_m = 0$

ρ	ν	μ
0.10	16447 ± 2504	1.6 ± 1.45
0.25	5427 ± 447	0.0 ± 0.00
0.50	2742 ± 369	0.0 ± 0.00
1.00	1536 ± 192	0.0 ± 0.00
2.00	2173 ± 767	0.1 ± 0.26
4.00	8524 ± 1175	3.6 ± 0.96

$\rho_m = 0.5$

ρ	ν	μ
0.05	5091 ± 538	0.0 ± 0.00
0.10	2411 ± 310	0.0 ± 0.00
0.25	1310 ± 404	0.0 ± 0.00
0.50	1353 ± 796	0.2 ± 0.00
1.00	2968 ± 1309	0.7 ± 0.77

$\rho_m = 0.9$

Table 4.6: Rumelhart, et al.'s Modified Algorithm on the Multiplexer Task

The output value of a semilinear unit is a real value between 0 and 1. To compare with the other algorithms that use binary-valued, linear threshold units as the output unit, the output of output unit k is set to 1 if $y_k \geq 0.5$ and is otherwise set to 0 while calculating μ and ν and the learning curve. This is only done in measuring performance, so the behavior of the learning algorithm is not altered.

From Table 4.5 we see that Rumelhart's algorithm reliably solved the multiplexer task within 100,000 steps, for $\rho = 0.1$ and 0.25 and $\rho_m = 0.9$. For $\rho = 0.25$ only 1,747 errors were accumulated over 100,000 steps ($\mu = 1,747$). Best performance (considering both ν and μ) resulted when $\rho = 0.25$ and $\rho_m = 0.9$. These parameter values were used to generate the learning curve of Figure 4.3, averaged over 30 runs of 300,000 steps each. The curve shows that extremely good performance is achieved very early in the runs; as early as 6,000 steps the average number of errors per step is below 0.06. The performance measures associated with this learning curve are $\nu = 0.00 \pm 0.00$ and $\mu = 1,962 \pm 148$.

The third curve in Figure 4.3 is from an experiment designed to test the modification to Rumelhart, et al.'s algorithm proposed by Sutton (1985). He suggests that the sign of the weight value appearing in the expression in Step 5 above is the important contribution of the weight, and that the magnitude of it might hamper the algorithm's progress, particularly when the magnitude is very small. We tested this hypothesis by replacing w_{jk} with the sign of w_{jk}, resulting in a new expression for $\delta_j[t]$:

$$\delta_j[t] = \left(\sum_{k \in O} \delta_k[t] \, \mathrm{sgn}(w_{jk}[t]) \right) \, y_j[t] \, (1 - y_j[t]).$$

As before, we varied ρ, with the results shown in Table 4.6 which are averaged over 10 runs of 100,000 steps each. The best value of ρ is still 0.25 and for ρ_m it is 0.9. The results averaged over 30 runs of 300,000 steps, using these parameter values, are $\nu = 0.00 \pm 0.00$ and $\mu = 1,354 \pm 575$, and the learning curve is shown in Figure 4.3. The modification appears to retard the algorithm's initial progress, but it still reliably solves the task, and results in a cumulative error measure, μ, not significantly different from that of the unmodified Rumelhart algorithm. The modified algorithm does appear to be more robust than Rumelhart, et al.'s unmodified algorithm; the task is reliably solved ($\nu = 0.00$) for a wider range of parameter values.

4.3 Reinforcement Learning

Four reinforcement-learning algorithms were studied, two being variants of one of the others. As reviewed in Chapter III, Barto, et al., have developed and studied several classes of reinforcement-learning algorithms (Barto, 1985; Barto and Anandan, 1985; Barto and Sutton, 1981a; Sutton, 1984). Sutton (1984) empirically compared a number of these algorithms. For tasks most similar to those faced by hidden units in the systems applied to the multiplexer task, Sutton found that a particular algorithm, which we will call "associative search with reinforcement prediction", or AS-RP, performed better than others.

4.3.1 Associative Search with Reinforcement Prediction

The AS-RP algorithm employs an additional unit that adjusts its weights, v, so as to match as closely as possible the value of the reinforcement received for each input vector. This provides the hidden units with a "reference" signal to which the current reinforcement can be compared to determine whether it is greater or less than the reinforcement usually received when given the current input vector. This extra unit performs a prediction of the reinforcement when given an input vector. The AS-RP algorithm is defined as follows:

Associative Search with Reinforcement Prediction (AS-RP)

1. Initialize all weights to zero:

$$\begin{aligned} w_h &= 0, \\ w_o &= 0, \\ v &= 0. \end{aligned}$$

2. Receive input vector, calculate output of all units, and receive error signals for the output units. The output, y_j, of hidden unit j is given by:

$$y_j[t] = \begin{cases} 1, & \text{if} \sum_{i=0}^{6} w_{ij}[t]\, x_i[t] + \eta_j[t] > 0; \\ 0, & \text{otherwise,} \end{cases}$$

   where the $\eta_j[t]$ are sequences of random variables with density function

$$f(s) = \frac{1}{1 + e^{-s}}.$$

3. Apply the perceptron learning algorithm to the output units.

4. Calculate the global reinforcement signal for the hidden units:

$$r[t] = 1 - \frac{1}{m} \sum_{j \in O} |d_j[t] - y_j[t]|,$$

   where m is the number of output units. Since $m = 1$, $r[t] \in \{0, 1\}$.

5. Calculate the prediction of reinforcement, r_p, as follows:

$$r_p[t] = \sum_{i=1}^{n} v_i[t]\, x_i[t],$$

   where n is the number of input components to the system and $v_i[t]$ is the predictor-unit's weight associated with input component $x_i[t]$.

ρ	ν	μ
0.01	23.7 ± 3.56	$121,756 \pm 131$
0.04	11.6 ± 4.70	$95,602 \pm 8,825$
0.16	2.8 ± 3.12	$42,149 \pm 16,354$
0.64	4.6 ± 4.86	$46,453 \pm 21,080$
1.28	12.1 ± 5.56	$75,454 \pm 20,789$

$\rho_p = 0.01$

ρ	ν	μ
0.01	24.3 ± 3.51	$121,867 \pm 224$
0.04	10.2 ± 5.29	$90,639 \pm 14,970$
0.16	5.7 ± 3.11	$61,708 \pm 15,251$
0.64	9.3 ± 4.13	$65,222 \pm 16,051$
1.28	5.4 ± 4.18	$40,169 \pm 20,829$

$\rho_p = 0.03$

ρ	ν	μ
0.01	24.6 ± 2.78	$121,436 \pm 270$
0.04	13.7 ± 4.61	$95,407 \pm 8,819$
0.16	3.5 ± 2.04	$48,320 \pm 13,120$
0.64	4.7 ± 3.87	$50,817 \pm 24,206$
1.28	13.2 ± 4.59	$79,975 \pm 21,263$

$\rho_p = 0.1$

Table 4.7: Associative Search with Reinforcement Prediction on the Multiplexer Task

6. Apply the associative search rule to hidden unit j:

$$\Delta w_{ij}[t] = \rho \left(r[t] - r_p[t]\right) \left(y_j[t] - \pi_j[t]\right) x_i[t],$$

where

$$\pi_j[t] = E\left\{y_j[t] | w_j[t]; x[t]\right\},$$

which is the expected value of the output y_j of unit j, given its current weight values and input. Since $y_j \in \{0, 1\}$, π_j is the probability that $y_j = 1$.

7. Update the predictor's weights.

$$\Delta v_i[t] = \rho_p \left(r[t] - r_p[t]\right) x_i[t].$$

8. Repeat, starting with Step 2, until the prespecified number of time steps have elapsed.

Two parameters control this algorithm: the rate of change in modifying the hidden units' weights is ρ, and the rate of change of the reinforcement predictor's weights is ρ_p. Five values of ρ were tried while ρ_p was set to one of three values. For each set of parameter values, 10 runs were made of 300,000 steps each.

The results in Table 4.7 show that the AS-RP algorithm did not solve the multiplexer task, but for $\rho = 0.16$ and $\rho_p = 0.01$ the value of ν was about 2.8, meaning that after 300,000 steps an average of only 2.8 out of 64 input vectors resulted in an incorrect output. The performance of the AS-RP algorithm over time is shown by its learning curve in Figure 4.4. The learning curve is averaged over 30 runs using $\rho = 0.16$ and $\rho_p = 0.01$, and resulted in $\nu = 3.36 \pm 1.98$ and $\mu = 48,754 \pm 8,662$. Better performance might be realized by testing additional parameter values.

Another possible way to improve this algorithm's performance is to include the output of the hidden units in the set of input components to the reinforcement-predictor unit. It may be impossible for a single unit to implement an accurate mapping from input vectors to reinforcement values, just as it is impossible for a single unit to implement a multiplexer function. This possibility was not tested.

4.3.2 Associative Reward-Penalty

The second algorithm from the reinforcement-learning class that we studied is the Associative Reward-Penalty, or A_{R-P}, algorithm, defined by Barto and Anandan (1985; Barto, 1985). This

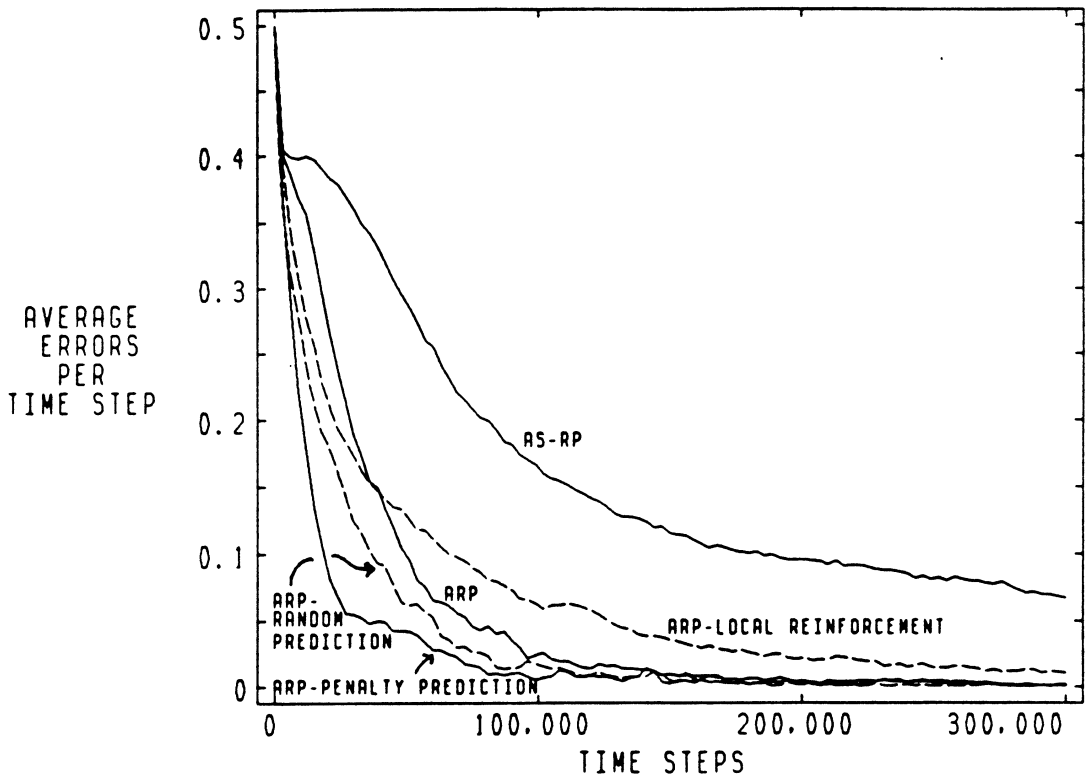


Figure 4.4: Learning Curve for Reinforcement Learning Algorithms on the Multiplexer Task

algorithm is an extension of the Linear Reward-Penalty algorithm from the learning automata literature (Narendra and Thathachar, 1974). Learning automata probabilistically search their output space and directly update their output probabilities as a function of the reinforcement received. In order to be sensitive to different input vectors, a different set of output probabilities must govern output selection for each input. Barto and Anandan's extension is the use of a unit's weights and input to determine the unit's output probabilities (as in the AS-RP algorithm). This algorithm is discussed more fully in Chapter III.

Associative Reward-Penalty (A_{R-P}) Algorithm

1. Initialize all weights to zero:
$$w_h = 0,$$
$$w_o = 0.$$

2. Receive input vector, calculate output of all units, and receive error signals for the output units. Output functions are those used for the AS-RP algorithm

3. Apply the perceptron learning algorithm to the output units.

4. Calculate the global reinforcement signal for the hidden units.
$$r[t] = 1 - \frac{1}{m} \sum_{j \in O} |d_j[t] - y_j[t]|$$
where m is the number of output units. For the multiplexer task, $m = 1$, so $r[t] \in \{0, 1\}$, but in general $r[t] = [0, 1]$.

5. Apply the A_{R-P} rule to each hidden unit j:
$$\begin{aligned} \Delta w_{ij}[t] \;=\; & \rho\, r[t]\, (y_j[t] - \pi_j[t])\, x_i[t] \\ & + \rho\, \lambda\, (1 - r[t])\, (1 - y_j[t] - \pi_j[t])\, x_i[t], \end{aligned}$$
where
$$\pi_j[t] = E\left\{ y_j[t] \,|\, w_j[t]; x[t] \right\},$$
This equation for the change in weight values is written for continuous r in the range [0,1], thus it is related to S-model learning automata (Narendra and Thathachar, 1974). Since $r \in \{0, 1\}$ for the multiplexer task, this equation could be expressed with two cases, for $r = 1$ and $r = 0$, as was done in Chapter III. Barto and Anandan's (1985) convergence proof for the A_{R-P} algorithm is only valid for $r \in \{0, 1\}$.

6. After the prespecified number of steps have elapsed the final-step performance measure ν is calculated.

The A_{R-P} algorithm depends on two parameters. The rate of weight change is controlled by ρ and λ. If $\lambda = 0$, no change is made to the weight values when the "penalty" signal $r[t] = 0$ is received, thus the algorithm is analogous to the reward-inaction scheme of learning automata. A symmetric reward-penalty rule results when $\lambda = 1$. For further comparisons see Barto (1985).

Table 4.8 contains the results of the A_{R-P} algorithm on the multiplexer task, averaged over 10 runs of 300,000 steps each. Of the parameter values tested, $\rho = 1$ and $\lambda = 0.004$ resulted in the best performance, solving the task with a final number of errors over all input vectors of 0.02.

The learning curve in Figure 4.4 shows that the A_{R-P} algorithm performed much better than the AS-RP. Averaged over 30 runs of 300,000 steps each, and using $\rho = 1$ and $\lambda = 0.004$, the A_{R-P} algorithm resulted in $\nu = 0.01 \pm 0.01$ and $\mu = 15,725 \pm 3,129$. The value of 0.01 ± 0.01 for ν indicates that the solution to the multiplexer task was reliably found.

λ	ν	μ
0.001	0.52 ± 1.30	$24,960 \pm 10,347$
0.002	0.33 ± 0.80	$25,234 \pm 7,486$
0.004	0.02 ± 0.01	$14,493 \pm 5,557$
0.008	0.01 ± 0.01	$20,046 \pm 5,918$
0.016	18.80 ± 6.65	$107,729 \pm 7,101$
0.032	23.00 ± 3.26	$118,806 \pm 185$

$\rho = 1$

ρ	ν	μ
0.1	1.30 ± 2.02	$51,109 \pm 10,157$
0.2	0.08 ± 0.03	$25,377 \pm 5,479$
0.4	1.80 ± 4.63	$20,301 \pm 7,050$
0.8	0.71 ± 1.53	$17,320 \pm 7,553$
1.0	0.02 ± 0.01	$14,493 \pm 5,557$
1.6	0.01 ± 0.00	$15,167 \pm 3,908$
3.2	11.60 ± 8.63	$81,798 \pm 16,013$

$\lambda = 0.004$

Table 4.8: A_{R-P} Algorithm on the Multiplexer Task

4.3.3 Local Reinforcement

The AS-RP and the A_{R-P} algorithms function in a "global" reinforcement paradigm, where every hidden unit is given the same numerical reinforcement signal. Every unit is searching for weight values that maximize the same evaluation function. However, as hidden units in a multilayer system, the application of these reinforcement-learning algorithms could be provided with more information than the global reinforcement signal. We investigated one possible way of using this information to construct a unique "local" reinforcement signal to each hidden unit. The approach is similar to Rosenblatt's back-propagation algorithm in its division into several cases according to units' outputs and weight values, but differs in that reinforcements are propagated rather than errors.

A_{R-P} *with Local Reinforcement*

Steps 1 through 3 are identical to those of the A_{R-P} algorithm.

4. Let $r_{jk}[t]$ be a reinforcement based on the output value of output unit k and the weight connecting hidden unit j to output unit k, defined as:

$$r_{jk}[t] = \begin{cases} 0.5, & \text{if } w_{jk}[t] = 0; \\ 1, & \text{if } w_{jk}[t] \neq 0 \text{ and } d_k[t] = y_k[t] \\ & \quad \text{or} \\ & y_j[t] = 1 \text{ and } w_{jk}[t](d_k[t] - y_k[t]) > 0; \\ & \quad \text{or} \\ & y_j[t] = 0 \text{ and } w_{jk}[t](d_k[t] - y_k[t]) < 0; \\ 0, & \text{otherwise.} \end{cases}$$

Calculate the local reinforcement signal for hidden unit j as:

$$r_j = \prod_{k \in O} r_{jk}[t],$$

though, for this task $O = \{5\}$, so $r_j = r_{j,5}$.

5. Apply the A_{R-P} rule to the hidden units, now using unique reinforcements for each:

$$\begin{aligned} \Delta w_{ij}[t] \;=\; & \rho\, r_j[t]\, (y_j[t] - \pi_j[t])\, x_i[t] \\ & + \lambda\, \rho\, (1 - r_j[t])\, (1 - y_j[t] - \pi_j[t])\, x_i[t]. \end{aligned}$$

6. After the prespecified number of steps have elapsed the final-step performance measure ν is calculated.

λ	ν	μ
0.0	1.32 ± 2.24	$16,069 \pm 10,750$
0.00001	3.07 ± 2.84	$28,822 \pm 15,678$
0.0001	0.24 ± 0.52	$12,512 \pm 4,937$
0.0002	1.07 ± 1.72	$17,830 \pm 7,706$
0.0005	0.39 ± 0.55	$10,418 \pm 1,973$
0.001	0.76 ± 0.87	$14,539 \pm 1,997$
0.002	4.64 ± 5.51	$21,145 \pm 2,279$
0.004	6.53 ± 4.69	$34,506 \pm 1,613$
0.008	10.30 ± 3.18	$64,436 \pm 1,644$

$\rho = 1$

ρ	ν	μ
0.01	23.72 ± 3.05	$12,2127 \pm 116$
0.25	2.84 ± 3.61	$40,581 \pm 16,852$
0.50	0.55 ± 1.27	$17,109 \pm 7,221$
0.75	1.31 ± 1.69	$23,290 \pm 10,555$
1.00	0.24 ± 0.52	$12,512 \pm 4,937$
1.25	1.44 ± 2.66	$22,467 \pm 12,298$

$\lambda = 0.0001$

Table 4.9: A_{R-P} with Local Reinforcement on the Multiplexer Task

The motivations for the cases in Step 4 are as follows. When a hidden unit has no influence on the output unit, i.e., $w_{jk} = 0$, then no preference in its output should be revealed. To accomplish this, r_{jk} is set to 0.5 regardless of the output of the hidden unit, the output unit, and the correct output. The second case is composed of three situations. First, if the hidden unit does have a nonzero output weight, i.e., $w_{jk} \neq 0$, and the output unit generated a correct response, then the hidden unit is "rewarded" by being assigned a reinforcement value of 1, increasing the probability of the output value that it just produced. The second part rewards the hidden unit if its output value is 1 and its output weight has the same sign as the output unit's error. The third part rewards the unit when its output value is 0 and its output weight differs in sign from the output unit's error.

This modification to the A_{R-P} algorithm does not add any new parameters. We tried a number of values for ρ and λ and averaged the results over 10 runs of 300,000 steps each. From Table 4.9 we see that $\rho = 0.5$ and $\lambda = 0.0001$ resulted in the best value of ν, which was 0.55 errors over the 64 input vectors after 300,000 steps. The cumulative measure, μ, was lowest for $\lambda = 0.0005$.

A learning curve for the A_{R-P} with local reinforcement, again averaged over 30 runs of 300,000 steps each, is included in Figure 4.4. The values $\rho = 0.6$ and $\lambda = 0.0001$ were used for the algorithm's parameters. This modification to the A_{R-P} algorithm performs slightly better that the original A_{R-P} algorithm before approximately the $20,000^{\text{th}}$ step, and thereafter its performance is worse than that of the A_{R-P}.

The local reinforcement addition seems to help during the early stages, but is a hindrance throughout the remainder of a run. Perhaps this indicates that using the information about the hidden units' output weights and the output units' errors is only beneficial while the hidden units have minor effects on the output unit through output weights of small magnitudes. When output weights are near zero, learning according to the A_{R-P} algorithm is very slow, because there is very little correlation between a hidden unit's output and the global reinforcement signal. But as the output weights increase in magnitude they acquire more of an influence on the global reinforcement and can begin to optimize their weight values. A more complex task—one requiring more than a single output unit—might demonstrate a greater potential for using local reinforcements.

4.3.4 Penalty Prediction

The plight of hidden units that have not yet acquired, or have lost, a substantial influence on the output units is to learn very slowly if they are modifying their weights through efforts to increase the reinforcement value. Is there some way to put such "unused" units to better use? We applied a second extension of the A_{R-P} algorithm to the multiplexer task to investigate this question.

This extension is based on the assumption that poor performance is caused by the lack of an appropriate representation. Situations for which incorrect outputs are generated need to be represented differently, perhaps with additional components, giving the output units more degrees of freedom with which they can alter their outputs.

To realize this idea we divide the learning algorithm for the hidden units into two parts, each part coming into play at different stages. When a hidden unit has a substantial effect on the output units, then the normal A_{R-P} algorithm is followed. But when a hidden unit does not significantly affect the output units, the hidden unit adjusts its weights in an attempt to match them to input vectors that result in low reinforcement values, in effect becoming a "penalty predictor". In this way, new features are introduced that represent inputs for which the performance of the system is low. This is related to the data-directed method of Reilly, Cooper, and Elbaum (1982), who dedicate new hidden units whenever an error is encountered by their system.

The implementation of this strategy depends on a measure of the degree to which a hidden unit has an influence on the output units. We simply used the magnitude of the hidden unit's output weight as an indication of influence, though, as Klopf and Gose (1969) showed, other measures might lead to more accurate indicators of influence. The magnitude of a hidden unit's output weight is used as the dependent variable in a logistic function to produce a measure of influence whose value ranges from 0 to 1. Some method of combining the measures from different output weights must be employed when the network has more than one output unit. The A_{R-P} algorithm is modified as follows:

A_{R-P} *algorithm with Penalty Prediction*

Steps 1 through 4 are identical to those of the A_{R-P} algorithm.

5. Apply the A_{R-P} algorithm with penalty prediction to the hidden units:

   (a) Calculate the influence, α_j, of hidden unit j on the output units:

$$\alpha_j[t] = \frac{1}{1 + e^{\frac{w_{jk}[t] - w_\alpha}{\tau_\alpha}}}.$$

   (b) Update the weights:

$$\begin{aligned} \Delta w_{ij}[t] \;=\; & \alpha_j[t] \left(\begin{array}{l} \rho\, r[t]\, (y_j[t] - \pi_j[t])\, x_i[t] \\ +\lambda\, \rho\, (1 - r[t])\, (1 - y_j[t] - \pi_j[t])\, x_i[t] \end{array} \right) \\ & +(1 - \alpha_j[t])\, \rho_\alpha\, (1 - r[t] - \pi_j[t])\, x_i[t]. \end{aligned}$$

6. Identical to the A_{R-P} algorithm.

The equation in Step 5b is composed of two main parts. The first part is the expression for the A_{R-P} algorithm. Its contribution to the update of weights varies inversely with that of the novel, second part of the equation. The second part is only significant when α_j is small, meaning that Hidden Unit j has little influence on the output unit. It serves to push the weight vector in the direction of the current input vector when r is small and it pushes the weight vector away from the input vector when r is large.

In addition to the parameters ρ and λ of the A_{R-P} algorithm, this modification depends on the values of ρ_α, w_α, and τ_α, which have their strongest effect when α_j, the influence on the output units, is small. The variable α_j is a function of Unit j's output weights, defined in such a way as to scale its value between 0 and 1; $\alpha_j = 1$ when Unit j has a very strong influence on an output unit, and $\alpha_j = 0$ when it has no influence. The scaling function for α is controlled by the parameters w_α and τ_α, and its form is that of the logistic function, where τ_α is the "spread" of the function and w_α is the value of its argument such that $\alpha_j = 0.5$ when $w_\alpha = w_{jk}[t]$. For example, if $w_\alpha = 1.5$ and $\tau_\alpha = 0.1$ and there is one output unit, then α_j will have the following values for the given values of Unit j's output weight:

output weight (w_{jk})	α_j
0	0.000
± 1	0.007
± 2	0.993
± 3	1.000

ρ_α	w_α	ν	μ
0	0.5	7.30 ± 5.06	$11,246 \pm 1,965$
1	0.5	23.50 ± 3.11	$20,009 \pm 223$
2	0.5	19.90 ± 6.85	$18,004 \pm 2,032$
4	0.5	4.70 ± 5.51	$10,795 \pm 3,324$
8	0.5	0.54 ± 1.21	$4,682 \pm 1,498$
16	0.5	3.20 ± 3.06	$6,143 \pm 2,389$
32	0.5	7.10 ± 5.29	$7,712 \pm 2,734$
4	1.0	22.60 ± 3.35	$19,786 \pm 325$
8	1.0	6.20 ± 6.75	$10,142 \pm 4,136$
16	1.0	0.37 ± 0.52	$6,779 \pm 1,626$
32	1.0	3.80 ± 3.85	$8,253 \pm 1,723$
4	1.5	14.70 ± 7.79	$17,297 \pm 2,781$
8	1.5	4.20 ± 5.51	$8,301 \pm 2,403$
16	1.5	0.26 ± 0.52	$4,761 \pm 2,225$
32	1.5	5.20 ± 5.52	$8,140 \pm 3,714$
4	2.0	25.00 ± 3.66	$20,209 \pm 53$
8	2.0	19.40 ± 6.28	$19,185 \pm 1,477$
16	2.0	12.70 ± 6.68	$14,733 \pm 4,688$
32	2.0	4.00 ± 3.84	$10,030 \pm 3,328$
4	4.0	23.80 ± 1.32	$20,234 \pm 55$
8	4.0	22.10 ± 3.19	$20,252 \pm 66$
16	4.0	23.70 ± 3.12	$20,254 \pm 66$
32	4.0	23.00 ± 3.21	$20,240 \pm 72$

$\rho = 1.0,\ \lambda = 0.004$
$\tau_\alpha = 0.1$

τ_α	ν	μ
0.01	2.78 ± 2.66	$7,695 \pm 2,304$
0.1	0.26 ± 0.52	$4,761 \pm 2,225$
0.2	0.48 ± 1.05	$8,698 \pm 3,030$
0.4	18.00 ± 5.89	$18,542 \pm 2,108$
0.6	23.80 ± 2.90	$20,242 \pm 100$
0.8	23.20 ± 2.27	$20,340 \pm 64$
1.0	23.70 ± 2.13	$20,243 \pm 74$

$\rho = 1.0,\ \lambda = 0.004$
$\rho_\alpha = 0.1,\ w_\alpha = 1.5$

Table 4.10: A_{R-P} with Penalty Prediction on the Multiplexer Task

For the multiplexer task, the output unit learns under the perceptron learning algorithm with a learning constant of $\rho = 1$. Therefore, the output unit's weights, which are the output weights of the hidden units, will always be integer-valued. For $w_\alpha = 1.5$ and $\tau_\alpha = 0.1$, the A_{R-P} with penalty prediction algorithm will approximate the original A_{R-P} algorithm except when the value of the output weight is 0, as it is initially, or 1.

Table 4.10 shows the results of testing this algorithm for various parameter values over 10 runs of 50,000 steps each. Using the values $\rho = 1$ and $\lambda = 0.004$, which gave the best performance for the original A_{R-P} algorithm, we found that $\rho_\alpha = 16$, $w_\alpha = 1.5$, and $\tau_\alpha = 0.1$ were the best parameter values tested. Not reported here are further experiments in which ρ and λ are varied, again finding $\rho = 1$ and $\lambda = 0.004$ to be the best of a small set of alternative values.

The best parameters were used to generate the learning curve in Figure 4.4, averaged over 30 runs of 300,000 steps each. The learning curve shows that this algorithm performed much better than the original A_{R-P}. The A_{R-P} with penalty prediction resulted in performance measures of $\nu = 0.08 \pm 0.17$ and $\mu = 7,411 \pm 1,773$. Roughly twice as many errors on average were made during the runs of the A_{R-P} algorithm ($\mu = 7,411$ versus $\mu = 15,725$).

The fact that large values of ρ_α result in better performance than small values suggests that the advantage of the penalty prediction addition is due to the size of the large jumps in a hidden unit's weights when the unit has a small output weight, and not in the direction of the weight change. This hypothesis was tested through further experiments, as follows.

The algorithm was modified in a way that preserved the size of the large weight changes while removing the dependence on the reinforcement signal to direct the weight change. Instead, a random signal guided the changes in weight values when the output weight is of low magnitude. Thus, Step 5b of the A_{R-P} with penalty prediction becomes

$$\begin{aligned}\Delta w_{ij}[t] \;=\; & \alpha_j[t] \left(\begin{array}{c} \rho\, r[t]\, (y_j[t] - \pi_j[t])\, x_i[t] \\ +\lambda\, \rho\, (1 - r[t])\, (1 - y_j[t] - \pi_j[t])\, x_i[t] \end{array} \right) \\ & +(1 - \alpha_j[t])\, \rho_\alpha (s_j[t] - \pi_j[t])\, x_i[t],\end{aligned}$$

where the $s_j[t]$ are sequences of Bernoulli random variables (possible values are 0 and 1).

The learning curve for the A_{R-P} with random prediction is shown in Figure 4.4. It's performance is worse than that of the A_{R-P} with penalty-prediction algorithm, suggesting that there is an advantage in predicting penalties. However, it performs better than the simple A_{R-P} algorithm. Thus, there is also an advantage to taking undirected, large steps in the search for weight values for unused units. The increase in performance of the A_{R-P} with penalty prediction algorithm over the simple A_{R-P} is probably due to both effects.

4.4 Summary

To facilitate the comparison of the learning algorithms' performance on the multiplexer task, most of the learning curves are superimposed in Figure 4.5. Recall that the errors per time step are plotted by averaging over 30 runs and over bins of 3,000 step intervals. A non-learning, random strategy of selecting outputs would result in an average of 0.5 errors per time step.

It is easily seen that the classes of algorithms in order of decreasing performance are

1. error back-propagation (excluding Rosenblatt's algorithm),
2. reinforcement learning, and
3. direct search.

This ranking is supported by the values of the performance measures, shown in Table 4.11, where the algorithms are ranked according to their resulting values of μ. There is no statistically-significant difference between the values of μ for the two versions of Rumelhart's algorithm. However, the difference between these algorithms and the best reinforcement-learning algorithm, the A_{R-P} with penalty prediction, is significant.

Among the reinforcement-learning algorithms, some differences in μ are significant, while others are not. In particular, the results of the AS-RP algorithm are significantly worse than all other reinforcement-learning results. As discussed earlier, another version of the AS-RP algorithm should be tested: the output of the hidden units should be included as input to the reinforcement predictor, thus not restricting the reinforcement prediction to be a linear function of the input as originally represented.

All other differences are significant. The direct search algorithms are significantly worse than others, with the exception of Rosenblatt's algorithm, and their relative ranking is also significant.

In Chapter II, one possible set of new features was proposed that could be formed by a single layer of hidden units and would also provide for the existence of a solution to the multiplexer task. Now we can ask what new terms actually developed during successful runs of multilayer learning algorithms.

For a partial answer to this question, we analyzed two runs, one with Rumelhart, et al.'s algorithm and the other with the A_{R-P} with penalty prediction algorithm. Each run was interrupted at three points to determine the features that the hidden units had acquired at various stages. Figure 4.5 shows that a single run using Rumelhart's algorithm is very likely to have solved the multiplexer task by the $10,000^{\text{th}}$ step, so the run was analyzed after 2,000, 5,000, and 10,000 steps. The results of this analysis appear in Table 4.12. A unit's state is specified by a logical expression for the union of all input vectors for which the output of the unit is 1. For example, Unit 1 on the $2,000^{\text{th}}$ step responds with output 1 for input vectors $(0,0,0,0,0,1)^T$ and $(0,0,0,1,0,1)^T$ (disregarding the constant component of the input vectors). Labeling the components of the input vectors as $(a_1, a_2, d_1, d_2, d_3, d_4)$, for address lines a_1, a_2 and data lines d_1, d_2, d_3, d_4. The expression for the union of these vectors is $\bar{a}_1\bar{a}_2\bar{d}_1\bar{d}_3d_4$. Included with each hidden unit expression is the approximate value of the unit's output weight, indicating how it affects the output value of the output unit.

In addition to the hidden-unit analysis, expressions were determined for the output unit, both with and without the features generated by the hidden units. Let us start our discussion of Table 4.12 with these expressions, by first studying the last row. At step 2,000, a relatively complex expression developed for the output unit, but by step 10,000 the unit's expression is

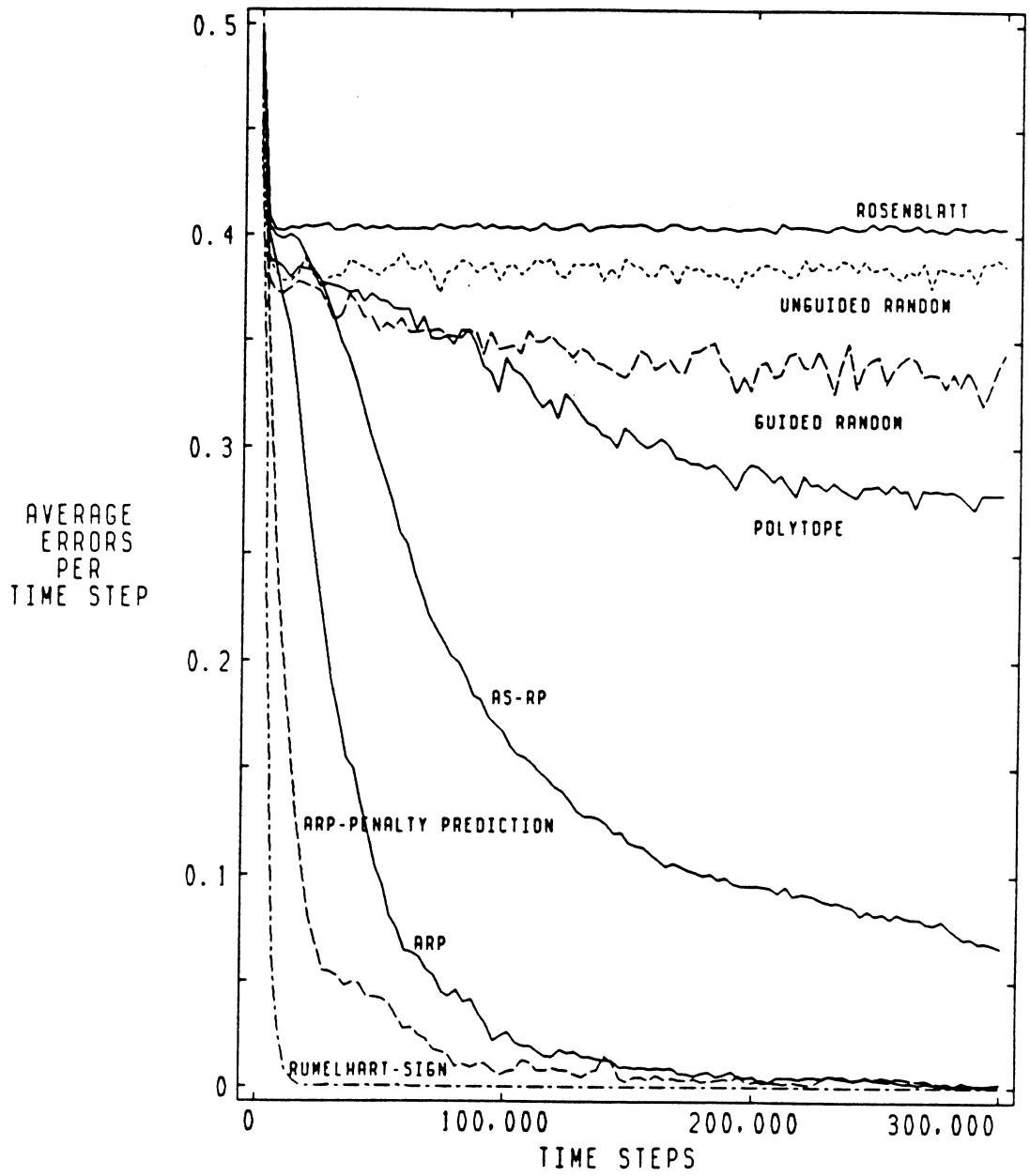


Figure 4.5: Learning Curves for All Algorithms on the Multiplexer Task

Algorithm	ν	μ	parameters
Rumelhart sign of output weight	0.00 ± 0.00	$1,354 \pm 575$	$\rho = 0.25,\ \rho_m = 0.9$
Rumelhart	0.00±0.00	$1,962 \pm 148$	$\rho = 0.25,\ \rho_m = 0.9$
A_{R-P} with penalty prediction	0.08±0.17	7,411±1,773	$\rho = 1,\ \lambda = 0.004,$ $\rho_\alpha = 16,\ w_\alpha = 1.5,\ \tau_\alpha = 0.1$
A_{R-P} with random prediction	0.01±0.00	10,695±2,690	$\rho = 1,\ \lambda = 0.004,$ $\rho_\alpha = 16,\ w_\alpha = 1.5,\ \tau_\alpha = 0.1$
A_{R-P}	0.01±0.01	15,725±3,129	$\rho = 1,\ \lambda = 0.004$
A_{R-P} with local reinforcement	0.65±1.08	20,467±6,923	$\rho = 0.6,\ \lambda = 0.0001$
AS-RP	3.36±1.98	48,754±8,662	$\rho = 0.16,\ \rho_p = 0.01$
polytope	14.2±2.09	94,977±3,079	$n = 1600,\ m = 10,$ $c_r = 2,\ c_e = 2,\ c_c = 0.2$
guided random	13.1±2.36	103,866±3,420	$n = 3200, \tau = 1$
unguided random	17.0±2.93	115,062±229	$n = 1600$
Rosenblatt	23.9±1.58	121,115±92	$\rho = 0.5,$ $p_1 = 0.9,\ p_2 = 0.3,\ p_3 = 0.1$

Table 4.11: Performance Summary for Multiplexer Task

	Step 2,000	Step 5,000	Step 10,000
Unit 1	$\bar{a}_1\bar{a}_2\bar{d}_1\bar{d}_3d_4$ (-2)	$\bar{a}_1\bar{a}_2(\bar{d}_1\bar{d}_3 \vee \bar{d}_1d_2d_4)\vee$ $a_1\bar{a}_2(\bar{d}_1d_2\bar{d}_3 \vee \bar{d}_1\bar{d}_3d_4)$ (-7)	$\bar{a}_1\bar{a}_2(\bar{d}_1\bar{d}_3 \vee \bar{d}_1d_4)\vee$ $a_1\bar{a}_2(\bar{d}_1\bar{d}_3 \vee \bar{d}_1d_4)$ (-7)
Unit 2	$\bar{a}_1\bar{a}_2\bar{d}_1\bar{d}_3d_4$ (-2)	$\bar{a}_1\bar{a}_2\bar{d}_1$ (-7)	$\bar{a}_1\bar{a}_2\bar{d}_1$ (-11)
Unit 3	*null* (-1)	$\bar{a}_1\bar{a}_2\bar{d}_2d_3d_4\vee$ $\bar{a}_1a_2(\bar{d}_2d_4 \vee \bar{d}_2d_3 \vee \bar{d}_1\bar{d}_2)$ (-9)	$\bar{a}_1a_2\bar{d}_2$ (-12)
Unit 4	*null* (-1)	$\bar{a}_1\bar{a}_2(\bar{d}_1\bar{d}_3d_4 \vee d_2\bar{d}_3d_4)\vee$ $a_1\bar{a}_2(\bar{d}_1\bar{d}_3d_4 \vee d_2\bar{d}_3d_4)$ (-3)	$a_1\bar{a}_2\bar{d}_3$ (-7)
output unit *without* hidden units	$\bar{a}_1\bar{a}_2(d_3d_4 \vee d_2d_3 \vee d_2d_4 \vee d_1\bar{d}_2d_4 \vee d_1d_3\bar{d}_4 \vee d_1d_2\bar{d}_3)\vee$ $\bar{a}_1a_2(d_1d_2d_4 \vee d_1d_3d_4 \vee d_1d_2d_3 \vee d_2d_3d_4)\vee$ $a_1\bar{a}_2(d_3d_4 \vee d_2d_3 \vee d_2d_4)\vee$ $a_1a_2(d_1d_2d_4 \vee d_1d_3d_4 \vee d_1d_2d_3 \vee d_2d_3d_4)$	$\bar{a}_1\bar{a}_2\vee$ $\bar{a}_1a_2\vee$ $a_1\bar{a}_2\vee$ $a_1a_2(d_4 \vee d_2d_3)$	$\bar{a}_1\bar{a}_2\vee$ $\bar{a}_1a_2\vee$ $a_1\bar{a}_2\vee$ $a_1a_2d_4$
output unit *with* hidden units	$\bar{a}_1\bar{a}_2(d_1d_2d_4 \vee d_1d_3d_4 \vee d_2d_3)\vee$ $\bar{a}_1a_2(d_1d_2d_4 \vee d_1d_3d_4 \vee d_2d_3)\vee$ $a_1\bar{a}_2(d_1d_2d_4 \vee d_1d_3d_4 \vee d_2d_3)\vee$ $a_1a_2(d_1d_2d_4 \vee d_1d_3d_4 \vee d_1d_2d_3 \vee d_2d_3d_4)$	$\bar{a}_1\bar{a}_2d_1\vee$ $\bar{a}_1a_2d_2\vee$ $a_1\bar{a}_2(d_3 \vee d_1d_2\bar{d}_3)\vee$ $a_1a_2(d_4 \vee d_2d_3)$	$\bar{a}_1\bar{a}_2d_1\vee$ $\bar{a}_1a_2d_2\vee$ $a_1\bar{a}_2d_3\vee$ $a_1a_2d_4$

Table 4.12: New Features Developed by Rumelhart, et al.'s Algorithm

	Step 10,000	Step 20,000	Step 50,000
Unit 1	$\bar{a}_1\bar{a}_2d_1\bar{d}_2\bar{d}_4\vee$ $a_1\bar{a}_2d_1\bar{d}_2d_3\bar{d}_4\vee$ $a_1a_2(d_1\bar{d}_2\bar{d}_4$ $\vee d_1\bar{d}_2d_3\bar{d}_4)$ (0)	*null* (-1)	$\bar{a}_1\bar{a}_2\bar{d}_1d_2\bar{d}_4$ (-7)
Unit 2	$\bar{a}_1\bar{a}_2(\bar{d}_1\bar{d}_2d_3 \vee \bar{d}_1d_2)\vee$ $a_1\bar{a}_2\bar{d}_1d_2$ (-7)	$\bar{a}_1\bar{a}_2\bar{d}_1\vee$ $a_1\bar{a}_2\bar{d}_1d_2$ (-13)	$\bar{a}_1\bar{a}_2\bar{d}_1\vee$ $a_1\bar{a}_2\bar{d}_1d_2$ (-13)
Unit 3	$\bar{a}_1a_2d_4\vee$ $\bar{a}_1a_2\bar{d}_2d_4\vee$ $a_1\bar{a}_2\bar{d}_2d_4$ (-8)	$\bar{a}_1\bar{a}_2d_4\vee$ $\bar{a}_1a_2\bar{d}_2d_4\vee$ $a_1\bar{a}_2(\bar{d}_2d_4 \vee d_2\bar{d}_3d_4)$ (-10)	$\bar{a}_1\bar{a}_2d_4\vee$ $\bar{a}_1a_2\bar{d}_2d_4\vee$ $a_1\bar{a}_2\bar{d}_2d_4$ (-16)
Unit 4	*null* (-1)	$a_1\bar{a}_2\bar{d}_3$ (-9)	$a_1\bar{a}_2\bar{d}_3$ (-25)
output unit *without* hidden units	$\bar{a}_1\bar{a}_2\vee$ $\bar{a}_1a_2(d_2d_3 \vee d_4)\vee$ $a_1\bar{a}_2(d_4 \vee d_3\bar{d}_4)\vee$ $a_1a_2d_4$	$\bar{a}_1\bar{a}_2\vee$ $\bar{a}_1a_2d_4\vee$ $a_1\bar{a}_2\vee$ $a_1a_2d_4$	$\bar{a}_1\bar{a}_2\vee$ $\bar{a}_1a_2(d_3 \vee \bar{d}_2d_4)\vee$ $a_1\bar{a}_2\vee$ $a_1a_2d_4$
output unit *with* hidden unit	$\bar{a}_1\bar{a}_2(d_1 \vee \bar{d}_1\bar{d}_2\bar{d}_3)\vee$ $\bar{a}_1a_2(d_2d_3 \vee d_2d_4)\vee$ $a_1\bar{a}_2(\bar{d}_2d_3 \vee d_2d_3d_4$ $\vee d_1d_2d_4 \vee d_1d_2d_3)$	$\bar{a}_1\bar{a}_2d_1\vee$ $\bar{a}_1a_2d_2d_4\vee$ $a_1\bar{a}_2d_3\vee$ $a_1a_2d_4$	$\bar{a}_1\bar{a}_2d_1\vee$ $\bar{a}_1a_2d_2\vee$ $a_1\bar{a}_2d_3\vee$ $a_1a_2d_4$

Table 4.13: New Features Developed by A_{R-P} with Penalty Prediction Algorithm

exactly the multiplexer expression, as expected. The expressions for the output unit without hidden units show that at step 10,000 the new features learned by the hidden units are necessary for the generation of the correct output for input vectors containing three of the four possible addresses; for address (1,1) the output unit itself is capable of producing the correct output.

Given the expression for the output unit without hidden units at step 10,000, it is clear how the terms formed by Units 2, 3, and 4 are being used. All have negative influences on the output unit, effectively carving out of the outputs unit's expression those input vectors for which the output unit produces a 1 when the correct output is 0. The role played by Unit 1 is much less clear, and would require a careful analysis of exact weight values for us to understand.

Table 4.13 shows the results of a similar analysis of a run with the A_{R-P} with penalty prediction algorithm. The run was interrupted at 10,000, 20,000, and 50,000 steps, a larger total number of steps than was used for the analysis of Rumelhart's algorithm. The expression for the output unit with hidden units at the $50,000^{\text{th}}$ step is indeed the multiplexer expression.

The manner in which the hidden units interact with the output unit to produce the correct output is not as straightforward as it was for the previous example. It is clear that Unit 4 has acquired the same role here as it did in the other run. This is purely a matter of coincidence, since there is no a priori bias among the hidden units; initially, each unit is equally likely to develop a particular feature.

In conclusion, this chapter presents the results of several multilayer learning algorithms on

a task that requires new features in order to solve it. The task is of sufficient scale that direct search methods perform poorly, never solving the task within the allotted time. Several error back-propagation algorithms were studied, of which Rumelhart, et al.'s algorithm readily dealt with the difficulties of the task, reliably solving it within several thousand steps. Reinforcement-learning algorithms were also tested, with various degrees of success. A particular contribution of this study is the proposal and demonstration of a new reinforcement-learning algorithm for which unused units attempt to predict the occurrence of low reinforcements, thereby expanding the representation of inputs for which the system performs poorly. Reinforcement-learning and a data-directed form of new-feature generation are combined into one learning algorithm.

The use of data-directed methods for the generation of new features is not limited to reinforcement-learning algorithms. Chapter III reviews several data-directed ways in which new features are generated for the correction of errors. A possibly fruitful research topic would be the combination of such data-directed methods with the exact gradient-descent method of Rumelhart, et al., (1986). Gradient-descent techniques can get stuck at local minima, but the generation of new features related to inputs for which the system produces incorrect responses might provide the extra degrees of freedom needed by the system to escape from the local minima.

In comparing the performance results of this chapter, it is important to keep in mind two critical limitations of this study. The most obvious limitation is that a single task was used. The results provide no indication of how the relative ranking of the algorithms would change if different tasks, either simpler or more complex, are used. Answers to the question of how well the algorithms scale-up to harder tasks require further experiments on tasks of varying complexity. Hand in hand with this issue is the issue of how an algorithm's performance is affected by altering the network architecture, such as the addition or removal of hidden units. Neither issue was investigated by this study.

The second limitation is due to the manner in which the values for each algorithm's parameters were chosen. The experimenter became part of every learning algorithm by trying a number of different parameter values. The values that resulted in the best performance for a particular algorithm were used in its comparison with the other algorithms. The time required to perform this parameter optimization process is not taken into account by the performance measures used in this study. An algorithm might rank very well according to the performance measures but be very sensitive to its parameter values and require much effort to find optimal parameter values. This does not appear to be the case for the results of this chapter. The algorithm with the best performance is Rumelhart's error back-propagation algorithm modified to use the sign of the output weight and it reliably solves the multiplexer task for a wide range of parameter values.

Chapter 5

Strategy Learning with Multilayer Connectionist Systems

The learning of strategies has been studied from a number of perspectives. Much of the research has concerned the development of existing strategies (either directly provided by an expert or extracted from sample solutions of an expert) into forms appropriate for a given task (see Keller [1982] for a review of this research). However, as stressed by Langley (1983), a system that learns from its own experience must possess additional capabilities for behavior generation and credit assignment based on the outcome of the behavior. The temporal credit-assignment problem complicates this kind of learning; there is no standard with which to compare the system's actions on every step.

This problem severely restricts the use of connectionist systems for strategy learning because most connectionist learning algorithms require knowledge of correct actions for a training set of input vectors. Learning proceeds by the presentation of input vectors from the training set and the modification of weights in a manner that is dependent on the error between the correct action and the actual action, as was done for the experiments of Chapter IV. This form of learning is called *supervised learning* since it is supervised by a teacher that knows the desired responses for the training set. If correct actions are known for certain problem states, supervised learning can be used to train a connectionist system to respond correctly for those states, but further learning from experience requires a different type of learning.

An example of the kind of connectionist learning systems needed to learn strategies from experience is presented in this chapter and demonstrated in subsequent chapters. One connectionist network, called the *evaluation network*, learns an evaluation function by a method that is based on Sutton's (1984) AHC algorithm. Simultaneously, a reinforcement-learning algorithm is used in a second network, the *action network*, to learn search heuristics as a probabilistic mapping from states to actions. This approach was taken by Barto, Sutton, and Anderson (1983; see also Sutton, 1984), but the networks there consisted of single units, limiting the complexity of evaluation functions and search heuristics that could be learned for a given representation of the problem's state. In this chapter, the capability of the system is increased by extending the learning algorithms to multilayer networks, thus decreasing the effort needed to design a problem-state representation.

5.1 Strategy Learning Behavior of the Algorithms

5.1.1 Initial Search Strategy

The initial search strategy followed by the action network is an unguided, random search, defined by the stochastic output functions of the output units. Action probabilities are used to select an

action for immediate application and the resulting heuristic reinforcement affects the modification of action probabilities. This is most appropriate when a model of the problem is not available, in which case transitions from the current state corresponding to different actions cannot be evaluated and a mechanism for selecting a single action based only on the current state (and perhaps past states) is required. For some problems models can be formed, and when the state space is not prohibitively large, systematic searches like breadth-first and depth-first search can be performed much more efficiently than the initial random search performed by the network. However, our goal is to develop a connectionist learning paradigm that does not require a model.

The learning algorithms of the connectionist system are not restricted to learning from behavior generated randomly. Any other search technique could be used but would require additional mechanisms for integrating the search process with the heuristic advice generated as the evaluation network's output. With the approach described in this chapter, the manner in which the action probabilities are updated after every step interfaces naturally with the probabilistic search performed by the units. Although the search is initially random, it gradually gains direction as the learning algorithm biases the action probabilities in favor of the more successful actions. Thus, the determination of the action probabilities, based on the state of the problem, is the connectionist system's counterpart to symbolic rules for expressing heuristics. This connection is discussed further in relation to the experimental results of the following chapters.

5.1.2 Credit Assignment

Langley distinguishes two classes of credit-assignment methods. One is based on knowledge of solution paths.[1] The initial search strategy must guarantee that solution paths will be discovered. Search heuristics are altered to recommend actions that move the puzzle's state onto a solution path, and to recommend against taking actions that move the state off of a solution path. Large search spaces make this approach impractical—many actions must be tried before a complete solution path becomes available. The alternative is to learn during the search process, before the goal is achieved. Anzai and Simon (1979) refer to this as *learning by doing*.

In order to learn before the goal state has been reached, heuristics must be available for the assignment of credit to actions as they are generated. Langley lists the following three heuristics that are domain-independent, but are specific to the class of tasks for which solution paths of minimum length are desired. For example, for the Tower of Hanoi task of Chapter VII a minimum length path is desired from the initial state to the goal state. However, the following heuristics are not useful for avoidance tasks, like the pole-balancing task of Chapter VI, where longer paths are more desirable. The heuristics are:

1. Avoid loops.
2. Shorter paths are desirable.
3. Avoid dead ends.

When a previously-visited state is revisited, the last action is marked as undesirable. Obviously, visiting a state more than once results in a non-minimal solution path. Similarly, when two paths are discovered between two states, the shorter of the two is desirable. The third heuristic assigns blame to the previous action when a state is reached from which all actions have been deemed undesirable. Humans seem to follow these heuristics in solving the Tower of Hanoi puzzle (Anzai and Simon, 1979).

The following two additional heuristics, also from Langley, incorporate domain-specific information:

4. Do not violate legality constraints.
5. Seek improvement in the value of an evaluation function.

[1] This discussion focuses on tasks with *goal states* and for which *solution paths* are desired. Analogous statements can be made concerning tasks with *failure states* that are to be avoided.

In most problem-solving tasks, each action can only be applied to a subset of states. Such constraints on the legal actions for a state are often embodied in the process of action selection, but can also be formulated as credit-assignment heuristics.

The last heuristic actually encompasses a range of heuristics, differing in the type of evaluation function used. An evaluation function can be naively based on clues assumed to be of use in guiding the search toward the goal, or it can reliably rank every state, guaranteeing that of any pair of states the state closer to the goal has the higher evaluation. Such an ideal evaluation function would subsume the roles of the other heuristics. However, it is very difficult to construct a good evaluation function for nontrivial tasks. Korf (1985) notes that an evaluation function based on a sequence of subgoals that appears to lead to the goal will not work if the subgoals are not *serializable*, i.e., if a satisfied subgoal must be violated to satisfy a subsequent subgoal.

To alleviate this dilemma, methods for the learning of evaluation functions have been proposed. Some methods learn only from complete solution paths, while others permit learning during the search process—again *learning by doing*. Rendell's (1983) method for learning evaluation functions relies on the discovery of a complete solution path. An interesting aspect of his approach is the division of the state space (or feature space, if only certain features of a state are presented to the evaluation function) into regions, and the learning of a unique, linear evaluation function for each region. In this way, complex mappings from states to evaluations can be constructed.

Samuel's (1959) well-known system for learning to play checkers does not require a complete solution path, a practical impossibility for games with so many states and moves. He used a polynomial with variable coefficients as an evaluation function. The difference between the evaluation of a state and a *backed-up* evaluation from a state encountered during a short look-ahead search defines an error, which guides the modification of the polynomial's coefficients. Samuel's learning algorithm met with limited success—the single linear evaluation function is not sufficiently complex to represent an evaluation function that is useful for all phases of the game. Sutton's (1984) AHC algorithm is a generalization of Samuel's method and of others. It is also limited to linear functions, but the extension presented in this chapter to multiple layers of units relaxes this restriction. Hampson (1983) developed an algorithm for learning evaluation functions that is similar to Sutton's AHC algorithm, though its development is less rigorous. Hampson also combined his algorithm with a method for learning in hidden units and for learning in an action network.

5.1.3 Modification

Generally, search strategies are improved by adding heuristics in the form of symbolic rules that limit the number of alternatives for each search step. This can be accomplished either by creating heuristics for evaluating actions or states and using a best-first type of search, or by creating heuristics that directly indicate the best actions for a given state.

The learning of symbolic rules has received considerable attention in the machine learning literature. Langley (1985) reviews generalization-based methods (e.g., Winston, 1975) which start with very specific hypotheses about appropriate conditions on the rules and generalize the conditions as experience is gained. Opposed to these are the discrimination-based methods (e.g., Langley, 1982) which start with general conditions which are refined with experience. Mitchell's (1977) version space method is an alternative that both generalizes and refines conditions.

On the surface, the heuristics learned by connectionist systems are much different from those expressed as symbolic rules. The differences are not so much in what can be expressed, but in how the heuristics are modified. Most learning procedures for connectionist systems do not assume that the information available at each step is sufficient to determine exactly how the heuristics should be modified. The information from each step is used to make only minor adjustments, and as additional experience is gained the cumulative adjustments result in a large overall change. Thus, heuristics are not significantly transformed to account for the outcome of each search step, but are incrementally adjusted. Fewer assumptions are made with this approach: the state representation and evaluation might be noisy, there might be an infinite number of states, and actions might not have reliable effects. For these reasons, the connectionist approach usually requires many more steps than do symbolic approaches to learn solutions to the same tasks. In making such comparisons, it is important to note the above differences in the assumptions of the two approaches.

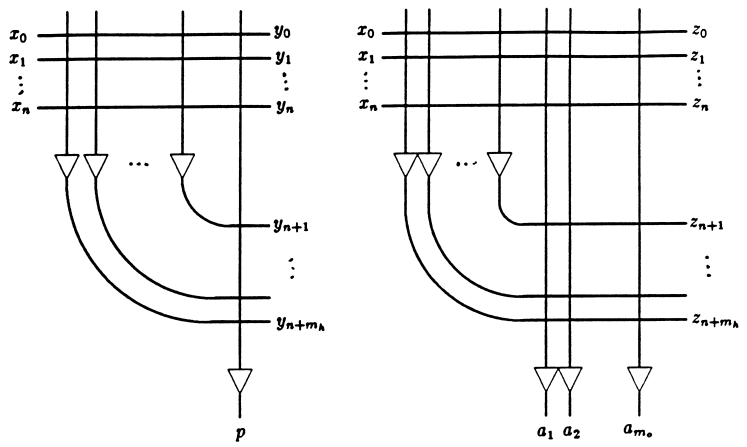


Figure 5.1: Two-Layer Networks for Strategy Learning

5.2 Connectionist Algorithms for Strategy Learning

As presented, the connectionist networks are two-layered, but the algorithms are easily extended to additional layers. The evaluation network and action network do not necessarily have the same number of hidden or output units, but since the particular network being discussed is always obvious from the context, the same variables are used to index units in both networks. Let there be m_h hidden units and m_o output units, for a total of $m = m_h + m_o$ units. The hidden units are indexed from 1 to m_h and the output units are indexed from $m_h + 1$ to m. (The evaluation network has only a single output unit.) Let H and O respectively denote the sets of hidden units and the output units. The evaluation and action networks do not share hidden units in order to avoid the difficulties of integrating the hidden-unit learning algorithms for the evaluation network and for the action network.

This structure is shown in Figure 5.1. The triangles represent the "computation-center" of the units; values of input components arrive from the left and "pass through" their weighted connection points, represented by intersections of horizontal and vertical lines, and down to the units, where an output is computed and sent out the output lines emanating from the apex of the triangles. Input from the environment is represented by $x_0, x_1, \ldots, x_n$ and is provided to all hidden units and output units. There is an interconnection weight at every intersection—hidden units receive $n + 1$ inputs and have $n + 1$ weights each, while output units receive $n + 1 + m_h$ inputs and have $n + 1 + m_h$ weights. The weights of the evaluation network are labeled v_{ij} for the i^{th} input to Unit j, and the analogous weight of the action network is w_{ij}. Particular values of these variables are referenced by the corresponding time step, e.g., $x_1[t]$, $v_{ij}[t]$, and $w_{ij}[t]$.

The learning algorithm for the evaluation network is composed of Sutton's (1984) AHC algorithm for the output unit and Rumelhart, Hinton, and William's (1986) error back-propagation algorithm for the hidden units. The AHC algorithm results in a prediction of future reinforcement for a given state. Changes in this prediction are used as *heuristic reinforcement* to guide the learning of search heuristics by the action network. The output units of the action network follow a reinforcement-learning algorithm[2] also studied by Sutton (1984; and Barto, Sutton, and Anderson, 1983). It is combined with the error back-propagation algorithm for hidden units. The connectionist learning system described in this chapter and demonstrated in Chapters VI and VII is based on the work of Barto, Sutton, and Anderson (1983) who demonstrated a very similar system without the hidden units.

[2]Other reinforcement-learning algorithms, such as the A_{R-P} algorithm, could be used for the output unit. The A_{R-P} algorithm would require an additional mechanism for scaling the heuristic reinforcement to be between 0 and 1. We chose to draw on our previous experience with the single-layer connectionist system (Barto, Sutton, and Anderson, 1983) by employing the reinforcement-learning algorithm used there.

5.2.1 Output Functions

Evaluation Network

The output of the evaluation network is computed in the following way. First, the outputs of the hidden units are calculated. The output, p_j, of Hidden Unit j is calculated using the values of its weights, v, at time t_v and input, x, at time t_x, as follows:

$$p_j[t_x, t_v] = f\left(\sum_{i=0}^{n} x_i[t_x]\, v_{i,j}[t_v]\right), \qquad \text{for } j \in H,$$

where f is the following logistic function:

$$f(s) = \frac{1}{1+e^{-s}}.$$

Variables representing the output of units of the evaluation network are doubly time-indexed to permit the multiplication of weight and input vectors from different time steps, required by the learning algorithms for reasons described in the next section.

The input vector, y, for the output unit is composed of the input from the environment and the output of the hidden units:

$$y_i[t_x, \cdot] = x_i[t_x], \qquad \text{for } i = 0, \ldots, n,$$

$$y_i[t_x, t_v] = p_{i-n}[t_x, t_v], \quad \text{for } i = n+1, \ldots, n+m_h.$$

The index, m, of the single output unit is dropped from p_m for clarity. Thus, the output of the evaluation network is p, and is defined as

$$p[t_x, t_v] = \sum_{i=0}^{n+m_h} y_i[t_x, t_v]\, v_{i,m}[t_v].$$

Action Network

To define the output of the action network we first define the hidden unit outputs, a_j:

$$a_j[t] = f\left(\sum_{i=0}^{n} x_i[t]\, w_{i,j}[t]\right), \qquad \text{for } j \in H.$$

These values partly determine the input vector, z, for the output units, along with the input from the environment:

$$z_i[t] = x_i[t], \qquad \text{for } i = 0, \ldots, n,$$

$$z_i[t] = a_{i-n}[t], \quad \text{for } i = n+1, \ldots, n+m_h.$$

For the experiments in later chapters, the output components, $a_j,\ j \in O$, of the action network are in one-to-one correspondence with the possible actions defined for the task.[3] To select an

[3]Rather than having actions and output units in one-to-one correspondence, each action can be encoded by a pattern of output values from a number of output units. For example, the six possible actions for the Tower of Hanoi puzzle could be represented as patterns of output values over three output units. This can lead to generalization among actions represented by similar output patterns, which can either benefit or hinder the learning of correct actions. These issues are not addressed by the one-to-one representation described in the text, although the reinforcement-learning algorithm is capable of dealing with the credit-assignment problem that results when output patterns encode actions.

action for a given problem state, the output of one output unit is set to 1 and the outputs of other units are set to 0 by the following process. The output functions of the reinforcement-learning units are stochastic, i.e., their output depends on a noisy weighted sum of inputs. A competition among the output units is implemented by assigning the value 1 to the unit with the highest weighted sum plus noise. This competition is limited to units corresponding to legal actions for the current state. Let $L_t \subset O$ be the set of indices for the output units that represent legal actions for the state at time t. The determination of L_t at each time step can be implemented by a network and even learned through experience, though for our experiments we specified L_t a priori. The responses of the output units are calculated as follows.

Let s_j be the noisy weighted sum of the input for Unit j, $j \in L_t$, defined as

$$s_j[t] = \sum_{i=0}^{n+m_h} z_i[t]\, w_{i,j}[t] + \eta_j[t],$$

where $\eta_j[t]$ is a sequence of random variables from the probability distribution Ψ, i.e., $\Psi(q) = P\{\eta_j \leq q\}$. or the pole-balancing task of Chapter VI, Ψ is defined as:

$$\Psi(q) = \frac{1}{1 + e^{-q}}.$$

The unit with the largest value for s_j wins the competition and is assigned a nonzero output:

$$a_j[t] = \begin{cases} 1, & \text{if } s_j[t] > s_k[t], \quad \text{for } k \in L_t \text{ and } j \neq k; \\ \\ 0, & \text{otherwise.} \end{cases}$$

To simplify the determination of the unit with the largest s_j for the Tower of Hanoi task, the following exponential probability distribution is used for Ψ:

$$\Psi(q) = 1 - e^{-q}.$$

The output function can be simplified for tasks with only two possible actions for every state, such as the pole-balancing task of Chapter VI. A single output unit is used whose binary output values encode the two actions. Let this unit be Unit k, i.e., $O = \{k\}$. The specialization of the output function for this case is:

$$a_k[t] = \begin{cases} 1, & \text{if } s_k[t] > 0; \\ \\ 0, & \text{otherwise,} \end{cases}$$

A weighted sum of 0 results in equal probabilities for generating the two output values, i.e., $P\{a_k[t] = 1\} = P\{a_k[t] = 0\} = 1/2$. As the weighted sum increases, $P\{a_k[t] = 1\}$ approaches 1, and as the weighted sum decreases $P\{a_k[t] = 1\}$ approaches 0, and $P\{a_k[t] = 0\}$ approaches 1.

5.2.2 Learning Algorithms

Output Layer of Evaluation Network

The change in p plus the value of the external reinforcement r, if present, is called the heuristic reinforcement, or $\hat{r}$, generally defined as:

$$\hat{r}[t] = r[t] + \gamma p[t, t-1] - p[t-1, t-1],$$

where $0 \leq \gamma < 1$, called the *discount rate*. The use of $\hat{r}$ in updating the weights of the evaluation network's output unit is meant to result in a prediction of future discounted reinforcement for the current state, with reinforcement further in the future discounted more than earlier reinforcement

(a formal analysis of this algorithm is currently being conducted by Sutton). States for which p is relatively large are favorable, while those with relatively low p are to be avoided. Once this mapping is correctly formed, changes in p can be used to indicate whether recent actions are leading towards favorable or unfavorable states.

The double time dependencies of variables in the equations for the evaluation network are needed for the following reason. In comparing one value of p with a previous value, care must be taken to avoid instability in the growth of weight values (equations for changing weight values are presented shortly). If the computation of p for step $t-1$ uses $v[t-1]$ whereas p for step t uses $v[t]$, then a change in p from one time step to the next could be caused by a change in weight values rather than the encounter of a state with a different expectation of reinforcement. To avoid this, the pair of subsequent p's is based on a single set of weight values, i.e., the difference between p for step $t-1$ and for step t is due only to the change from $x_i[t-1]$ to $x_i[t]$, because both p's are calculated using $v_i[t-1]$. If weights are known to change by small magnitudes on each step, then this precaution may not be necessary (as done in Barto, Sutton, and Anderson, 1983).

Sutton (1984) specialized the AHC algorithm by redefining $\hat{r}$ for several classes of tasks involving distinct *trials*, where a trial consists of the following steps:

1. setting the state of the problem to a *start state*,
2. letting the learning system and environment interact, until
3. a *goal state* or *failure state* is encountered, signaled by a particular external reinforcement value.

Following Sutton, $\hat{r}$ for trial-based tasks, like those of Chapters VI and VII, is defined to be:

$$\hat{r}[t] = \begin{cases} 0, & \text{if state at time } t \text{ is a start state;} \\ r[t] - p[t-1, t-1], & \text{if state at time } t \text{ is a goal or failure state;} \\ r[t] + \gamma\, p[t, t-1] - p[t-1, t-1], & \text{otherwise.} \end{cases}$$

The weights of the output unit, Unit m, of the evaluation network are updated by the following equation:

$$v_{i,m}[t] = v_{i,m}[t-1] + \beta\, \hat{r}[t]\, y_i[t-1, t-1],$$

for $i = 0, \ldots, n + m_h$ and $\beta > 0$. A positive change in state evaluations, indicated by a positive $\hat{r}$, results in an increase (decrease) in weight values proportional to the corresponding positive (negative) input values on the preceding steps. In this way, the evaluation of the preceding state is altered, effectively shifting evaluations to earlier states.

The above expression is a simplification of Sutton's algorithm: in the algorithm's general form, $y_i[t-1, t-1]$ is a trace of previous values of y_i, called an *eligibility trace*. An example of an eligibility trace is a weighted average of past values of y_i with recent values weighted more heavily. This generally results in faster development of good evaluation functions. Eligibility traces can also be used in the weight update equations of the action network. We chose not to implement eligibility traces primarily for the following reason. Preliminary experiments with the pole-balancing task of Chapter VI showed that a single-layer action network functioning with eligibility traces and without an adaptive evaluation network, i.e., learning only from the external reinforcement, could learn to perform relatively well. However, our interests were in studying learning in hidden units, which are required for the development of a good evaluation function for the pole-balancing task as it is formulated in Chapter VI. We removed the eligibility traces from both networks to force a greater reliance on the evaluation function and to increase the number of failures early in a run, providing more external reinforcement and thus more opportunities to improve the evaluation function. Thus, our primary goal was not to achieve the fastest possible learning on this task but to investigate learning in hidden units.

Output Layer of Action Network

Output Unit j, $j \in L_t$, of the action network updates its weights according to:

$$w_{i,j}[t] = w_{i,j}[t-1] + \rho\,\hat{r}[t]\,\Big(a_j[t-1] - E\{a_j[t-1]|w;z\}\Big)\,z_i[t-1],$$

for $i = 0, \ldots, n+m_h$, where $E\{a_j[t-1]|w;z\}$ is the expected value of $a_j[t-1]$ conditional on the current values of w and z. Weight values are not changed for output units corresponding to illegal actions. The value of $a_j[t-1] - E\{a_j[t-1]|w;z\}$ can be viewed as a measure of the difference between action $a_j[t-1]$ and the action that is usually taken for the given values of $z_i[t-1]$ and $w_{ij}[t-1]$. Thus, the results of an unusual action have more of an impact on the adjustment of weights than do other actions. Since $a_j \in \{0,1\}$, the expected value of a_j is equal to the probability that a_j is 1, i.e.,

$$E\{a_j[t]|w;z\} = P\{a_j[t]=1\}.$$

Equations are derived for $P\{a_j[t]=1\}$ in Appendix B for the cases of two and three legal actions for the current state—the only cases that arise for the formulation of the Tower of Hanoi puzzle used in Chapter VII. The calculation of this probability is simplified for the pole-balancing task, since the binary-valued output of the single output unit encodes one of the two possible actions. In this case, $P\{a[t]=1\}$ is just $\Psi(q)$, where q is the unit's weighted sum of its input.

Hidden Layer of Evaluation Network

¿From the results of the comparative experiments of Chapter IV, we concluded that the error back-propagation algorithm of Rumelhart, Hinton, and Williams (1986) usually acquired missing features most rapidly (for the particular multiplexer task used in the experiments). However, this algorithm cannot be applied directly because it requires knowledge of the correct output; we do not know the correct action or the correct evaluation for a given state, which would be needed in order to calculate an error to be back propagated.

To apply an error back-propagation scheme to the hidden units of a network whose output layer is learning through reinforcements, a way of translating a reinforcement into an error must be found. This can be done in a heuristic manner by extracting from the reinforcement-learning equations the terms that govern weight updates in a fashion similar to the error terms in the gradient-descent rules. However, it is not obvious how to incorporate the eligibility traces often used in reinforcement-learning algorithms into a back-propagation scheme, which is another reason for not including traces for the experiments reported here. A formal analysis of the resulting algorithms and the degree to which they follow the gradient of the expected value of reward was not attempted.[4]

For the evaluation network, $\hat{r}$ plays the role of an error in the update of the output unit's weights. If $\hat{r}$ is positive, the unit's weights are altered to increase the network's output, p, for positive input; if $\hat{r}$ is negative, weights are altered to decrease the output. Therefore, we define the error of the output unit, δ^p_m, to be:

$$\delta^p_m[t-1] = \hat{r}[t],$$

where the superscript denotes the association with the evaluation network that generates output p. The error that is back-propagated from the output unit to Hidden Unit j is just $\hat{r}$, and Rumelhart, et al.'s (1986) expression with Sutton's (1985) modification for the error of Hidden Unit j, called δ^p_j, becomes:

$$\delta^p_j[t-1] = \delta^p_m[t-1]\,\mathrm{sgn}(v_{j+n,m}[t-1])\,y_j[t-1,t-1]\,(1-y_j[t-1,t-1]),$$

and their method for updating the hidden units' weights can be applied:

$$v_{i,j}[t] = v_{i,j}[t-1] + \beta_h\,\delta^p_j[t-1]\,x_i[t-1] + \beta_m\,\Delta v_{i,j}[t-1],$$

[4] Williams (1986) has proved that a similar algorithm, the $\mathrm{A_{R-I}}$ algorithm (Barto and Anandan, 1985), results in expected weight changes equal to the gradient of the expected value of the reinforcement.

for units $j \in H$ and inputs $i = 0, \ldots, n$. Note that the sign of Hidden Unit j's output weight rather than the weight value itself is used. This variation is used because the results of Chapter IV's comparative study suggest that the algorithm's sensitivity to the value of the learning rate parameter, here β_h, is decreased by the use of the sign of the weight.

Hidden Layer of Action Network

The equation for updating the weights of the action network's hidden units is a bit more complicated. Once $\hat{r}$ becomes a good evaluation of the previous action, the role of an error is played by the product of $\hat{r}$ and the difference between the previous action and its expected value. The sign of the product is an indication of whether the action probability should be increased or decreased. So the error in the output of Output Unit k, $k \in L_t$, of the action network is defined as:

$$\delta_k^a[t-1] = \hat{r}[t]\,(a_k[t-1] - E\{a_k[t-1]|w; z\}).$$

The back-propagated error to Hidden Unit j is used to compute the hidden unit's error:

$$\delta_j^a[t-1] = \sum_{k \in L_t} \left(\delta_k^a[t-1]\,\mathrm{sgn}(w_{j+n,k}[t-1])\right) z_j[t-1]\,(1 - z_j[t-1]),$$

and the weights are updated by the following equation:

$$w_{i,j}[t] = w_{i,j}[t-1] + \rho_h\,\delta_j^a[t-1]\,x_i[t-1] + \rho_m\,\Delta w_{i,j}[t-1],$$

for units $j \in H$ and inputs $i = 0, \ldots, n$. Disregarding the different errors that are back-propagated by the two networks, the learning algorithms applied to the hidden units of the two networks are identical. The sum over the products of output unit errors and weights is not included in the expression for a hidden unit's error in the evaluation network because there is only one output unit.

5.2.3 Parameters

The equations for the evaluation network are governed by the following parameters:

$$\begin{array}{rcll} \beta & = & \text{learning rate for the output unit} & (\beta > 0); \\ \beta_h & = & \text{learning rate for the hidden units} & (\beta_h > 0); \\ \beta_m & = & \text{momentum factor for the hidden units} & (\beta_m \geq 0); \\ \gamma & = & \text{discount rate} & (0 \leq \gamma < 1). \end{array}$$

Similar parameters appear in the equations for the action network:

$$\begin{array}{rcll} \rho & = & \text{learning rate for the output units} & (\rho > 0); \\ \rho_h & = & \text{learning rate for the hidden units} & (\rho_h > 0); \\ \rho_m & = & \text{momentum factor for the hidden units} & (\rho_m \geq 0). \end{array}$$

In applying this connectionist system to a task, it is important to test a number of values for each parameter to investigate the sensitivity of the algorithms with respect to the parameters. This was done for every experiment reported in this thesis.

Chapter 6

Learning a Solution to a Numerical Control Task

Tasks that are typically characterized numerically are easily expressed in a connectionist representation, although different representations will result in a wide range of learning abilities. In this chapter, we study a realistic, numerical control task, called the pole-balancing or inverted-pendulum task, which is difficult to solve for two reasons:

1. the evaluation function to be learned by the connectionist system cannot be formed by a single unit, and
2. a performance evaluation in the form of a failure signal appears only after a sequence of actions has been taken, making it difficult to identify which actions are good and which are bad.

The objective of these experiments is to show that multilayer connectionist systems can learn the solution to this type of numerical control task. Results show that hidden units learning by means of the error back-propagation algorithm can learn new features that are sufficient for the pole-balancing task to be solved. A single-layer network (without hidden units) is shown to be incapable of solving the task. The second difficulty is the lack of a performance evaluation until the end of an action sequence—an example of *delayed reinforcement*—making it difficult to determine which actions early in a sequence share the responsibility for a subsequent failure.

The connectionist system described in Chapter V deals with both of these difficulties. In applying this system of connectionist networks and learning algorithms to the pole-balancing task, very little knowledge about the task is assumed, only that failure is to be avoided. We show in Chapter VII how this results in a very general learning system whose algorithms and structures vary little between this numerical control task and the puzzle-solving task of Chapter VII.

6.1 The Pole-Balancing Task

The pole-balancing task involves a pole hinged to the top of a wheeled cart that travels along a track, as shown in Figure 6.1. Both pole and cart are constrained to movement in a plane, i.e., the environment is two-dimensional. The state at time t of this dynamical system is specified by four real-valued variables:

$$
\begin{aligned}
x_t &= \text{the horizontal position of the cart, relative to the track;} \\
\dot{x}_t &= \text{the horizontal velocity of the cart;} \\
\theta_t &= \text{the angle between the pole and vertical, clockwise being positive;} \\
\dot{\theta}_t &= \text{the angular velocity of the pole.}
\end{aligned}
$$

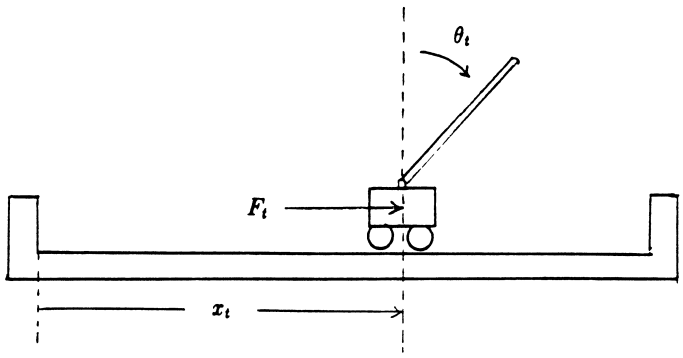


Figure 6.1: The Pole-Balancing Environment

The goal is to exert a sequence of forces, F_t, upon the cart's center of mass such that the pole is balanced for as long as possible and the cart does not hit the end of the track. More abstractly, the state of the cart-pole system must be kept out of certain regions of the state space, making this an avoidance control problem. There is no unique solution—any trajectory through the state space that does not pass through the regions to be avoided is acceptable. A minimal amount of knowledge about the task is assumed in our experiments. The only information regarding the goal of the task is provided by the external reinforcement signal, r_t, which signals the occurrence of a failure caused either by the pole falling past a prespecified angle, or the cart hitting the bounds of the track. r_t is defined as

$$r_t = \begin{cases} 0, & \text{if} -0.21 \text{ radians} < \theta_t < 0.21 \text{ radians and } -2.4 \text{ m} < x_t < 2.4 \text{ m}; \\ -1, & \text{otherwise.} \end{cases}$$

Note that r_t does not depend on $\dot{\theta}_t$ or $\dot{x}_t$.

The dynamics of the cart-pole system are given by the following equations of motion:

$$\ddot{\theta}_t = \frac{g \sin\theta_t + \cos\theta_t \left[\dfrac{-F_t - m_p l \dot{\theta}_t^2 \sin\theta_t + \mu_c \text{sgn}(\dot{x}_t)}{m_c + m_p} \right] - \dfrac{\mu_p \dot{\theta}_t}{m_p l}}{l \left[\dfrac{4}{3} - \dfrac{m_p \cos^2\theta_t}{m_c + m_p} \right]},$$

$$\ddot{x}_t = \frac{F_t + m_p l \left[\dot{\theta}_t^2 \sin\theta_t - \ddot{\theta}_t \cos\theta_t \right] - \mu_c \text{sgn}(\dot{x}_t),}{m_c + m_p}$$

where

m_c	=	1.0 kg	=	mass of the cart,
m_p	=	0.1 kg	=	mass of the pole,
l	=	0.5 m	=	distance from center of mass of pole to the pivot,
μ_c	=	0.0005	=	coefficient of friction of cart on track,
μ_p	=	0.000002	=	coefficient of friction of pivot,
g	=	-9.8 m/s^2	=	acceleration due to gravity.

A simulation of these dynamics was used in our experiments. Before describing the simulation and results, other approaches to the pole-balancing task are discussed.

6.2 Control-Engineering Approach

The pole-balancing task is frequently used to illustrate standard control techniques due to the inherent instability of the pole and the task's similarity to many balance-control problems. Cannon (1967) shows how the root-locus method is used to analyze the stability of a "lead compensation-network" controller that exerts a force proportional to the derivative of an error signal—in this case the pole's angular velocity. His analysis is confined to small angles and angular velocities and does not include the goal of avoiding the bounds of the track.

Eastwood (1968) used the pole-balancing task to contrast the calculus of variations and the dynamic programming approaches to optimal control. Both involve the optimization (minimization) of a performance index, defined over trajectories of x_t, θ_t, and F_t. The calculus of variations method and a quadratic performance index result in the linear control law:

$$F_t = ax_t + b\dot{x}_t + c\theta_t + d\dot{\theta}_t,$$

whereas the dynamic programming approach typically results in a computational procedure for determining F_t. Dynamic programming can deal with more realistic bounds on F_t, such as fixed maximum and minimum bounds.

These control-design techniques require a model of the system to be controlled in the form of differential equations that define how the state variables change over time. A good deal of time must be spent by the control engineer in determining a model that approximates the behavior of the system to the desired degree of accuracy. Control systems that learn without a predefined model, or that acquire internal models through observation of the system's behavior, would obviate this potentially difficult analysis.

6.3 Our Approach

An alternative to expressing a control law as an analytical equation is to represent the function in tabular form. Michie and Chambers (1968) took this approach for their learning system as applied to the pole-balancing task. Their table consisted of approximately 162 "boxes"—nonoverlapping, rectangular regions of the cart-pole system's state space—containing average counts of the number of steps before failure for a push to the right when the system's state addresses the corresponding box, and an analogous count for a push to the left. When a box is entered, the push with the highest count is applied. Their system successfully improved its performance with experience.

The learning system of Barto, Sutton, and Anderson (1983) integrated the table look-up approach with connectionist learning algorithms. Separate tables were used to store predictions of reinforcement and probabilities of generating actions, each indexed by the state of the cart-pole system. The tables are implemented as two units with linearly-weighted input components, each receiving 162 binary-valued input components. When the state is in a particular box, the corresponding input component is set to 1 and all other components are set to 0. Therefore, the weighted sum of the unit's input is equal to the value of the weight associated with the nonzero input component.

An obvious problem with the table look-up approach is that the size, shape, and placement of the regions into which the state space is divided greatly influence the ability of the system to learn the desired mappings. A region might be too large, meaning that different states inside the region require different output values, e.g., different pushes on the cart. Conversely, regions are smaller than optimal when many regions require the same output. If these regions are instead subsumed by one large region, then what is learned for one state is generalized correctly to all other states in the region. With many small regions, learning must occur in all regions independently. Michie and Chambers proposed a solution to this problem: regions for which one output is not clearly better than any other despite repeated experience should be "split" into several, smaller regions, and regions with the same output value should be "lumped" into a single, larger region. Politis and Licata (1986) have pursued this possibility with Barto, et al.'s, learning system and a technique for periodically splitting every region uniformly into a number of smaller regions.

The problem of selecting the best sizes and shapes of the regions is exactly the problem of designing the set of features for representing the state of the cart-pole system. In the following

experiments, the fixed "decoder" that Barto, et al. used to translate the cart-pole state to a region address is replaced by an additional adaptive layer of hidden units that learns features useful in solving the pole-balancing task.

This "adaptive decoder" view is closely related to current research topics in control theory involving the application of multiple controllers to one task. For example, the control of a full 360-degree pole requires a complex control law in order to be useful for all states. An alternative is to use a collection of less-complex control laws, and activate one at a time, based on the current state and an ordering of the control laws according to their ability to deal with that state. Learning when to switch from one controller to another is analogous to learning how to classify the state into one of a set of boxes.

Another example of connectionist systems applied to the pole-balancing task comes from the work of Widrow and Smith (1964). They present results of using a supervised-learning scheme to train a network of Adaline units (Widrow, 1962) to duplicate the responses of a teacher. For their experiments the teacher was a predefined linear control law. A human could play the role of the teacher by manually controlling the pole through an interface, such as a joystick, and the Adaline network could use the human's responses as training examples. Learning to mimic a teacher is much easier than learning from delayed reinforcement as is required for our formulation of the pole-balancing task.

6.4 Experiments

6.4.1 Simulation

The cart-pole system was simulated on a digital computer by numerically approximating the equations of motion using Euler's method with time step $\tau = 0.02$ seconds, resulting in the following state equations:

$$\begin{aligned} x_{t+\tau} &= x_t + \tau \dot{x}_t, \\ \dot{x}_{t+\tau} &= \dot{x}_t + \tau \ddot{x}_t, \\ \theta_{t+\tau} &= \theta_t + \tau \dot{\theta}_t, \\ \dot{\theta}_{t+\tau} &= \dot{\theta}_t + \tau \ddot{\theta}_t. \end{aligned}$$

For the ensuing discussion, discrete time is used by considering each time interval of length τ as a single time step. Letting state $(x_t, \dot{x}_t, \theta_t, \dot{\theta}_t)$ be indicated by $(x[t], \dot{x}[t], \theta[t], \dot{\theta}[t])$, the above equations become the following difference equations:

$$\begin{aligned} x[t+1] &= x[t] + \tau \dot{x}[t], \\ \dot{x}[t+1] &= \dot{x}[t] + \tau \ddot{x}[t], \\ \theta[t+1] &= \theta[t] + \tau \dot{\theta}[t], \\ \dot{\theta}[t+1] &= \dot{\theta}[t] + \tau \ddot{\theta}[t]. \end{aligned}$$

6.4.2 Desired Functions

From successful experiments with two-layer systems, we discovered that the desired evaluation and action functions are as sketched in Figure 6.2. For clarity, let us limit attention to projections of the functions to the $(\theta, \dot{\theta})$ subspace. Figure 6.2a shows the kind of function expected to be learned by the evaluation network. Failure is likely to occur in the upper right and lower left portions of the $(\theta, \dot{\theta})$ state space. We want the learning system to shift this failure signal to states that precede failure states, then to states that precede failure by longer time intervals, with the strength of the shifted prediction indicative of the average number of time steps until failure. Without this map, the system can only learn when the external failure signal arrives. Past states and actions, or weighted averages of previous states and actions, could be used to apportion blame for the failure to previous actions, but the tradeoff between a) the need for a long history to blame actions many steps in the past, and b) the need for a short history to avoid blame being spread too thinly (resulting in slow learning), is difficult to optimize. Learning this evaluation function

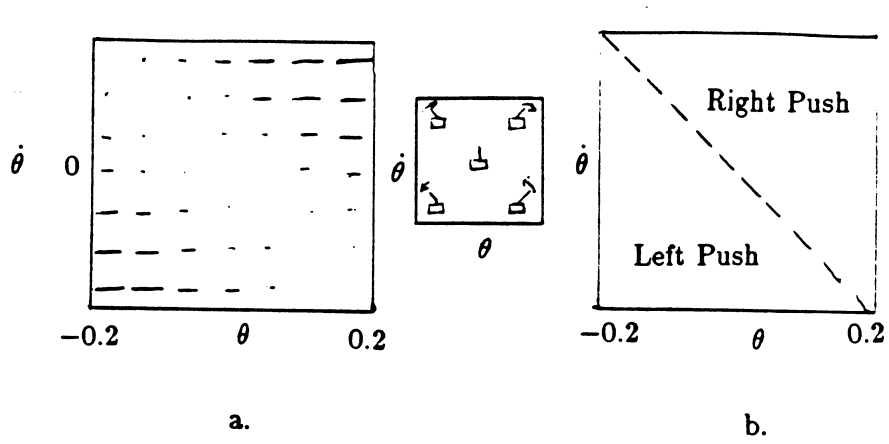


Figure 6.2: Desired Functions for Pole-Balancing Solution

allows actions very early in a sequence that ends in failure to be blamed if they are responsible for moving the cart-pole system to a state with a lower evaluation. These *temporal* credit-assignment issues are studied in detail by Sutton (1984), who developed the AHC algorithm used here in the output unit of the evaluation function.

The map from $(x, \dot{x})$ to a prediction of failure also looks like Figure 6.2a. In the lower left corner, the cart is moving to the left and is near the left border of the track, and in the upper right corner it is approaching the right border of the track.

Figure 6.2b shows an action function for generating a push on the cart. For small angles, such as $-0.21 < \theta < 0.21$ as used in our experiments, the surface that separates states requiring different actions, called the *switching surface*, can be linear. States in the upper right region require a push to the right, while states in the lower left require a left push. These are the only actions available—the system is unable to produce a zero force. Such a control system is referred to as a *bang-bang* controller—the sign of the applied force can be changed, but the magnitude of the force is constant. The linear surface is an approximation to the true nonlinear switching surface of the optimal bang-bang controller. The linear approximation works well for the small range of angles used in the experiments. The position and slope of the linear surface varies for different values of x and $\dot{x}$.

Knowledge of these desired functions was not used during learning. Obviously, any clues as to what the desired mappings are should be applied to the design of the learning system. For a connectionist system, this involves the initialization of weight values such that appropriate features are formed by the hidden units and the correct actions are generated by the output units. However, our objective was to study the learning ability of the connectionist system without such predefined knowledge to see how applicable the system may be to tasks for which this knowledge is not easily obtainable.

6.4.3 Interaction between Learning System and Cart-Pole Simulation

The components of the networks' input, $x_i[t]$, are scaled versions of the state variables:

$$\begin{aligned} x_1[t] &= \tfrac{1}{4.8}(x[t] + 2.4), \\ x_2[t] &= \tfrac{1}{3}(\dot{x}[t] + 1.5), \\ x_3[t] &= \tfrac{1}{0.42}(\theta[t] + 0.21), \\ x_4[t] &= \tfrac{1}{4}(\dot{\theta}[t] + 2). \end{aligned}$$

An additional input, $x_0[t]$, with a constant value of 0.5 provides a variable threshold. Inputs $x_1[t]$ and $x_3[t]$ range from 0 to 1, while $x_2[t]$ and $x_4[t]$ are primarily within the 0–1 range, but can fall outside these bounds. This scaling accomplishes two things. Since the learning algorithms involve the input terms, $x_i[t]$, as factors in the equations for updating weight values, terms with predominantly larger magnitudes will have a greater influence on learning than will other terms. To remove this bias all input terms are scaled to lie within the same range. Secondly, since the values of the state variables are centered at zero, and due to the linear nature of the network's units, the correct action for positive θ and $\dot{\theta}$ will transfer to negative θ and $\dot{\theta}$ in exactly the right way, i.e., the correct action for negative θ and $\dot{\theta}$ *is* the negative of the correct action for positive θ and $\dot{\theta}$ (see Figure 6.2b). If the state variables are used without scaling, these correct generalizations would make the task much easier, circumventing the need for hidden units in the action network. Thus, scaling the state variables allowed us to test the learning algorithm for hidden units.

The force exerted on the cart's center of mass at time t is given by:

$$F_t = \begin{cases} 10 \text{ nt}, & \text{if } a[t] = 1; \\ \\ -10 \text{ nt}, & \text{if } a[t] = 0, \end{cases}$$

where $a[t]$ is the binary-valued output of the action network at time t. The sampling rate of the cart-pole system's state and the rate at which control forces are applied are the same as the basic simulation rate, i.e., 50 hz..

The experiments consisted of a number of *trials*, each starting with the cart-pole system set to a state chosen at random, and ending with the appearance of the failure signal. A series of trials constitutes a *run*, with the first trial of a run starting with weights initialized to random values between -0.1 and 0.1.

A sample of the interaction between the learning system and the cart-pole simulation is presented in Table 6.1. It shows the cart-pole state, the networks' input, the action generated, $F[t]$, the state evaluation, $p[t]$, and the failure signal, $r[t]$, for the first 14 steps. The steps are from the beginning of a run, so the actions are mostly random and the evaluation is zero ($p = 0$) until a failure occurs. The two failures shown are due to θ exceeding -0.21 radians, indicated by x_3 approaching 0. (e.g., for the next state after $t = 9$, $x_3 \leq 0$, so the state was randomly reset for $t = 10$). We want the learning system to learn to generate actions, $F[t]$, that maximize the number of time steps between occurrences of $r[t] = -1$. The only information available to the system is given by the sequences $x_i[t], i = 0, \ldots, 4$ and $r[t]$.

6.4.4 Results of One-Layer Experiments

We experimented with single-layer networks (no hidden units) to obtain performance measures with which the performance of the two-layer system could be compared. The learning algorithms for the one-layer networks depend on the three parameters, ρ, β, and γ. The value of γ was fixed at 0.9, while different values of ρ and β were crudely optimized (simulation time prevented an accurate optimization) by performing 2 runs of 500,000 steps each for approximately 25 different

					Input				Output		Failure
t	$x[t]$	$\dot{x}[t]$	$\theta[t]$	$\dot{\theta}[t]$	$x_1[t]$	$x_2[t]$	$x_3[t]$	$x_4[t]$	$F[t]$	$p[t]$	$r[t]$
1	1.90	0.09	−0.01	−1.03	0.90	0.59	0.54	0.17	−10	0.00	0
2	1.91	−0.11	−0.03	−0.74	0.90	0.53	0.48	0.24	−10	0.00	0
3	1.90	0.09	−0.04	−1.04	0.90	0.46	0.43	0.32	10	0.00	0
4	1.91	0.29	−0.07	−1.35	0.90	0.53	0.39	0.24	10	0.00	0
5	1.91	0.09	−0.09	−1.08	0.90	0.60	0.34	0.16	−10	0.00	0
6	1.91	0.29	−0.11	−1.40	0.90	0.53	0.28	0.23	10	0.00	0
7	1.92	0.48	−0.14	−1.72	0.90	0.60	0.23	0.15	10	0.00	0
8	1.93	0.68	−0.18	−2.06	0.90	0.66	0.16	0.07	10	0.00	0
9	1.94	0.49	−0.22	−1.82	0.90	0.73	0.08	−0.01	−10	0.00	−1
10	−1.76	−0.52	−0.14	−0.41	0.14	0.26	0.18	0.48	10	−0.03	0
11	−1.77	−0.32	−0.15	−0.74	0.13	0.33	0.17	0.40	10	−0.03	0
12	−1.77	−0.12	−0.16	−1.08	0.13	0.39	0.15	0.31	10	−0.04	0
13	−1.77	0.08	−0.18	−1.42	0.13	0.46	0.12	0.23	10	−0.04	0
14	−1.77	−0.12	−0.21	−1.19	0.13	0.53	0.07	0.15	−10	−0.04	−1

Table 6.1: Sample Interaction of Learning System and Cart-Pole Environment

Run	Trials	Last Trial
1	33,977	14
2	61,888	4
3	24,795	16
4	22,717	130
5	28,324	28
6	15,218	100
7	31,594	15
8	44,903	9
9	16,115	72
10	26,402	14

Table 6.2: Results of One-Layer System

sets of parameter values. Two performance measures were used to select the best parameters. The number of trials, averaged over runs with one set of parameter values, provides a rough measure of performance over the length of a run. To judge how well the solution had been learned by the end of the run, the number of steps in the last trial, or the previous trial, whichever is larger, is averaged over all runs. In this way, an abnormally short final trial caused by the termination of a run on the $500,000^{\text{th}}$ step does not enter into the average of final trial lengths.

Performance did not vary considerably for the parameter values that were tested. The best parameter values were used to obtain a more statistically-significant result by performing 10 runs of 500,000 steps each. The following parameter values were used:

$$\begin{aligned} \beta &= 0.05, \\ \rho &= 0.5, \\ \gamma &= 0.9, \end{aligned}$$

resulting in the number of failures for the 10 runs shown in Table 6.2. The average number of trials for each run is approximately 30,593. In addition, the number of steps in the last trial is shown for each run. As explained above, this value is actually the larger of the last trial length and the previous trial length, in case the last trial had just begun when the run was terminated at step 500,000.

The number of steps per trial versus the number of trials is plotted in Figure 6.3. The plotted values are averages over the 10 runs and over bins of 100 trials, i.e., the trials for a run are grouped into intervals of 100 trials, the number of steps per trial is averaged for each interval, and the

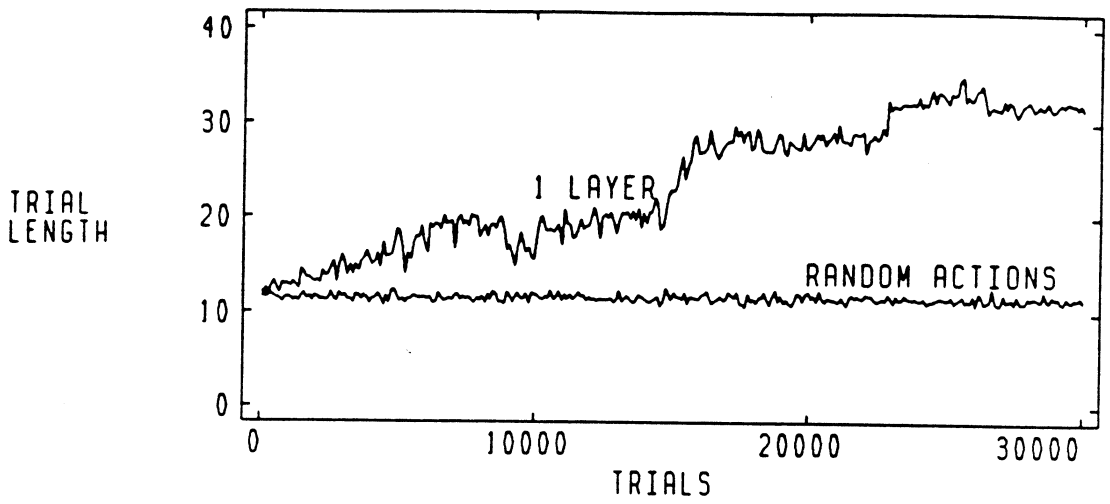


Figure 6.3: Balancing Time versus Trials for One-Layer System

results are averaged over the runs. The learning curve shows that performance does improve with experience; the trial length is approximately equal to 10 steps initially, and after 30,000 trials approximately 30 steps occur per trial. Even with little experience, the learning system performs better than a fixed controller that selects pushes on the cart at random. The large variance from trial to trial is due to the initialization of the cart-pole system to random states upon failure. The starting state of each trial might be very similar to a failure state, or it might be near the state of perfect balance where $(x, \dot{x}, \theta, \dot{\theta}) = (0, 0, 0, 0)$. This method of restarting after failure differs from that used by Barto, Sutton, and Anderson (1983), who started the cart-pole system at state (0,0,0,0) after every failure.

The values of the weights at the end of each run varied considerably. The best of the 10 runs, resulting in 15,218 failures, resulted in the weights that are displayed on the network schematic of Figure 6.4. Positively-valued weights are drawn as hollow circles and negative weights as filled disks. The magnitude of a weight is proportional to the radius of its circle, or disk. Appendix C contains a list of actual weight values. ¿From the size of the weights we see that the output of the evaluation network (Figure 6.4a) is rather insensitive to the values of the state variables, and the value of the output is always negative. The output of the action network (Figure 6.4b) does depend on the system's state. A large θ has a positive effect, producing a push on the cart to the right, and a large value for x has a negative effect, pushing the cart to the left.

A better understanding of what these weights mean is obtained from a graph of the output of the networks versus the state. To display these functions of four variables, we generated graphs of the functions' values versus θ and $\dot{\theta}$ for nine different pairs of x and $\dot{x}$ values. Figure 6.5a and Figure 6.5b contain such graphs for the evaluation network and the action network, respectively. The insensitivity of the evaluation network to the state is evident from the flat surfaces of its graphs. The base plane in these graphs does not represent a value of 0; the surface is actually at a small negative value. Obviously this function serves no useful role as an evaluation function for states—its value varies an insignificant amount from state to state. It is for this reason that the one-layer system could not improve its performance over 30 steps per trial. Credit is assigned by the external reinforcement signal only to actions that push the cart-pole into a failure state in one step. These actions may not be responsible for the failure and may even be correct for the state preceding failure.

Figure 6.5b shows that the action network has learned a function with approximately the desired shape (see Figure 6.2b). The height above the base plane represents the probability of generating a push to the right. The level of the base plane is at zero probability, so for states where the surface lies near the base plane a push to the left is generated with high probability.

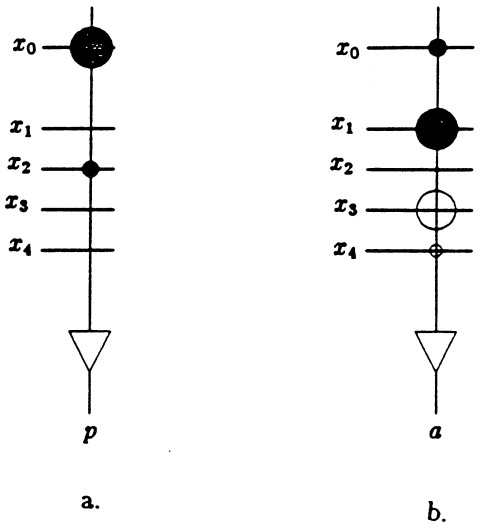


Figure 6.4: Weights Learned by One-Layer Network

The middle graph, where $x = 0$ and $\dot{x} = 0$, shows a smooth transition from a high probability of pushing left to a high probability of pushing right as θ and $\dot{\theta}$ go from negative to positive. This transition is shifted in the direction of negative θ for negative values of x and in the positive θ direction for positive values of x. In relating these graphs to those for the two-layer networks, we will see that this relationship between the transition line and the value of x is the opposite of what is needed to balance the pole while avoiding collisions with the track boundaries.

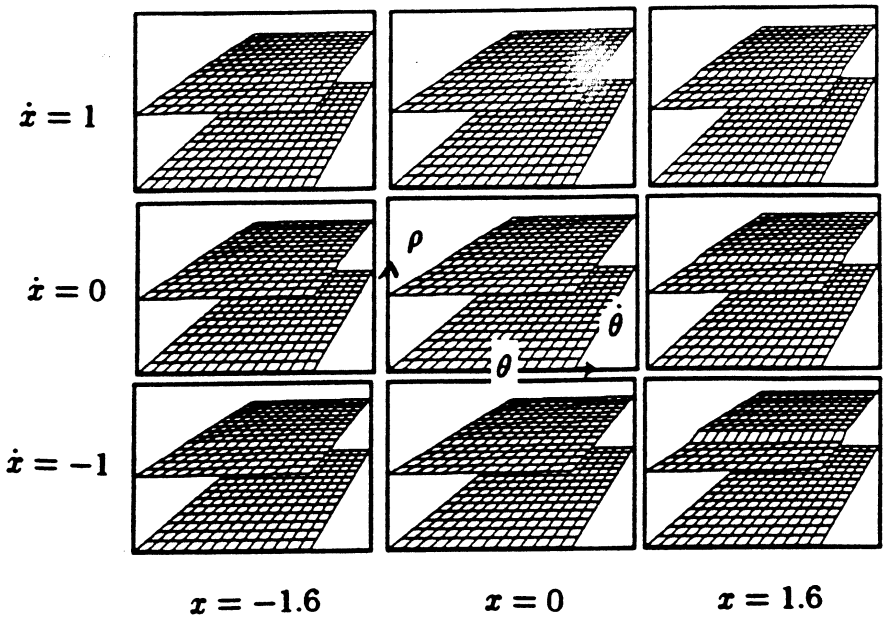
6.4.5 Results of Two-Layer Experiments

Two-layer networks were formed by adding 5 hidden units to the evaluation and action networks. The two-layer algorithms depend on the seven parameters, β, β_h, β_m, ρ, ρ_h, ρ_m, and γ. As was done for the one-layer system, sets of parameter values (approximately 10) were each tested in 5 runs of 500,000 steps. Performance varied significantly for small changes in parameter values (γ was not varied). The values giving the best performance are:

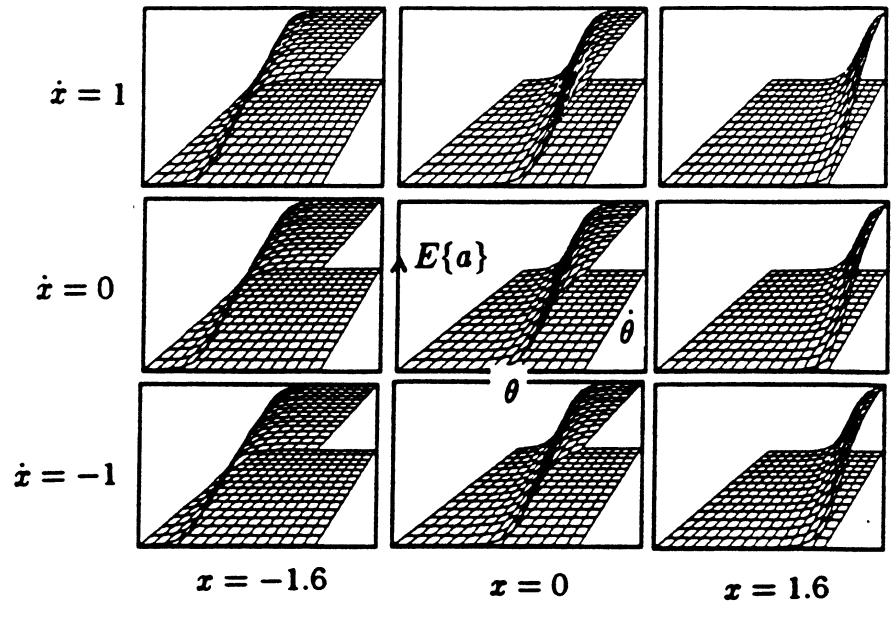
$$\begin{aligned} \beta &= 0.2, \\ \beta_h &= 0.2, \\ \beta_m &= 0, \\ \rho &= 0.5, \\ \rho_h &= 0.5, \\ \rho_m &= 0, \\ \gamma &= 0.9. \end{aligned}$$

Notice that $\beta_m = \rho_m = 0$. Results suggest that nonzero momentum terms in the hidden unit learning algorithms hinder performance on this task. These values were used for 10 runs of 500,000 steps, resulting in the total number of trials and final trial lengths shown in Table 6.3. The average number of trials over all runs is approximately 10,983, compared to 30,593 trials for the one-layer system. Even after much learning experience, a nonzero probability of selecting the wrong action exists for every state, as suggested by the relatively small number of steps in the last trials of Runs 1 and 10.

The learning curve for the two-layer system is shown in Figure 6.6. The large, stair-like jumps in the curve are due to the way in which performance is averaged over runs, described as follows.



a.



b.

Figure 6.5: Functions Learned by One-Layer Networks

Run	Trials	Last Trial
1	10,123	88
2	7,790	2,011
3	5,814	14,535
4	8,466	5,753
5	7,212	28,407
6	23,539	20,328
7	19,401	14,302
8	8,804	4,674
9	9,756	20,889
10	9,645	154

Table 6.3: Results of Two-Layer System

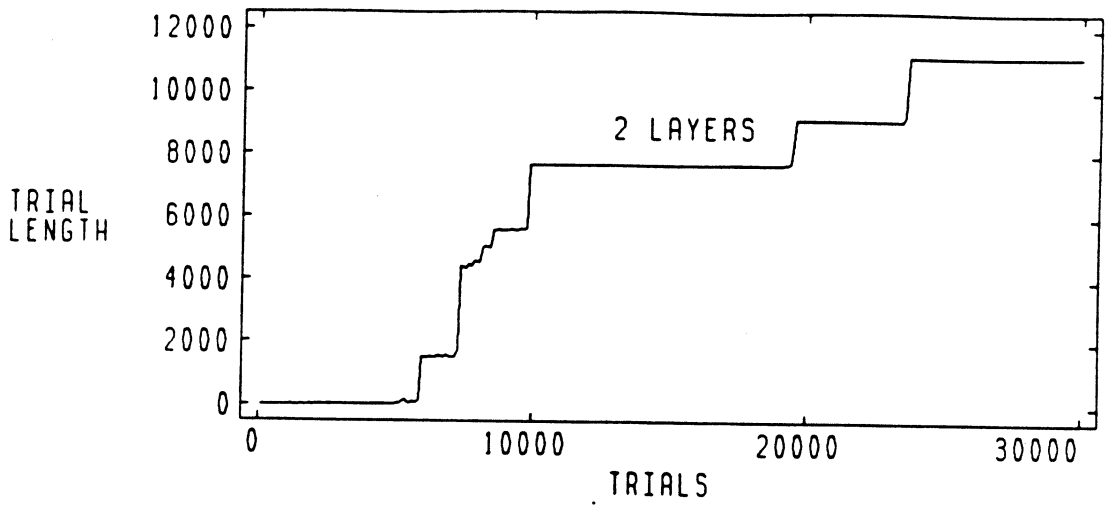


Figure 6.6: Balancing Time versus Trials for Two-Layer System

The axis for the number of trials is labeled from 0 to 30,000 trials, so runs for which less than 30,000 trials occurred were handled in a special way. The learning curve for such a run is extended to 30,000 trials by assigning to trials that didn't occur a value equal to the larger of a) the number of steps in the last trial, and b) the number of steps in the previous trial. In this way, a very short final trial is disregarded and the length of the previous trial is used to extend the curve. The large jumps in the curve occur when the last trial of a particular run is very long and the run is terminated when 500,000 steps have elapsed. If the experiments were run for more steps, the final performance level would be higher. This large number of steps is then averaged into the performance curve from that trial through Trial 30,000. For example, the jump at Trial 20,000 is due to the last trial of Run 7, which was approximately 14,000 steps in length. All 10 runs were terminated before 30,000 trials elapsed, resulting in a final performance level of about 11,000 steps per trial. Recall that the final level achieved by the one-layer system is only about 30 steps per trial.

The large number of weights, 35 in each network, makes it difficult to interpret the solutions found by the learning algorithms directly from the weight values. Their relative magnitudes and signs are shown in the network schematics of Figure 6.7. Figure 6.7a shows the final weight values for the evaluation network of Run 6, and Figure 6.7b shows the weights for the action network. Actual weight values are provided in Appendix C. Units 1, 2, 4 and 5 of the evaluation network are similar, having all positive weights. (In the figure, the small size of the corresponding circles make them appear to be filled-in disks.) Unit 3's weights differ, and it is also distinguished by having a large positive connection to the output unit. It appears that only Unit 3 has developed a new feature that is useful to the prediction of failure.

The function implemented by the evaluation network appears in Figure 6.8a. The height of the surface ranges from approximately -1.5 and 0.1. Its shape is just what is needed for the action network to receive an immediate evaluation of an action. At the center of each base plane, representing the $(\theta, \dot{\theta})$ subspace, the cart-pole is in a state where the pole is vertical and not falling. The evaluation has its highest value for these states, therefore forming an evaluation function that decreases as the cart-pole system moves away from this state. Any action that takes the system toward either the positive or negative $\theta, \dot{\theta}$ corner results in a negative evaluation change, i.e., a negative $\hat{r}$. The tilt of the surface as x and $\dot{x}$ change is also correct. Positive $\theta, \dot{\theta}$ states are more likely to result in failure when the cart is heading toward the right border of the track, where x and $\dot{x}$ are positive. Similarly, negative $\theta, \dot{\theta}$ states are likely to precede failure when the cart is heading to the left border, where x and $\dot{x}$ are negative.

Before discussing the features learned by the hidden units, let us look at the solution learned by the action network. From Figure 6.7b we see that again Hidden Unit 3 differs from the other units in its weight values: θ and $\dot{\theta}$ have large positive effects on Unit 3's output, and x and $\dot{x}$ have smaller positive and negative effects, respectively. Unit 3 is connected positively to the output unit, whereas the other units are connected negatively. The fact that the Unit 3's of both networks play significant roles is fortuitous; for other runs useful features are learned by a different set of units.

These hidden-unit influences in combination with the influences of the network's input and the weights on their direct connections to the output unit result in the action function displayed in Figure 6.8b. Two observations can be made in relation to the action function learned by the one-layer network, shown in Figure 6.5b. First, the transition from a high probability of pushing left to a high probability of pushing right is much quicker, as θ and $\dot{\theta}$ vary. This probability function implements a much more deterministic control than does that of the one-layer network. Due to the good evaluation function learned by the evaluation network, actions near the transition line are credited or blamed appropriately. A second observation is that the shift in the transition line as x and $\dot{x}$ vary is in the right direction. The pole should be balanced slightly to the right of vertical (positive θ) when the cart is near the left track boundary (negative x), and to the left of vertical when near the right boundary, resulting in a net action over several steps of a push towards the center of the track. To see that this is indeed what happens, note that the point at which the pole is balanced is roughly indicated by the location of the transition line. This line shifts toward positive θ when x is negative, and toward negative θ when x is positive.

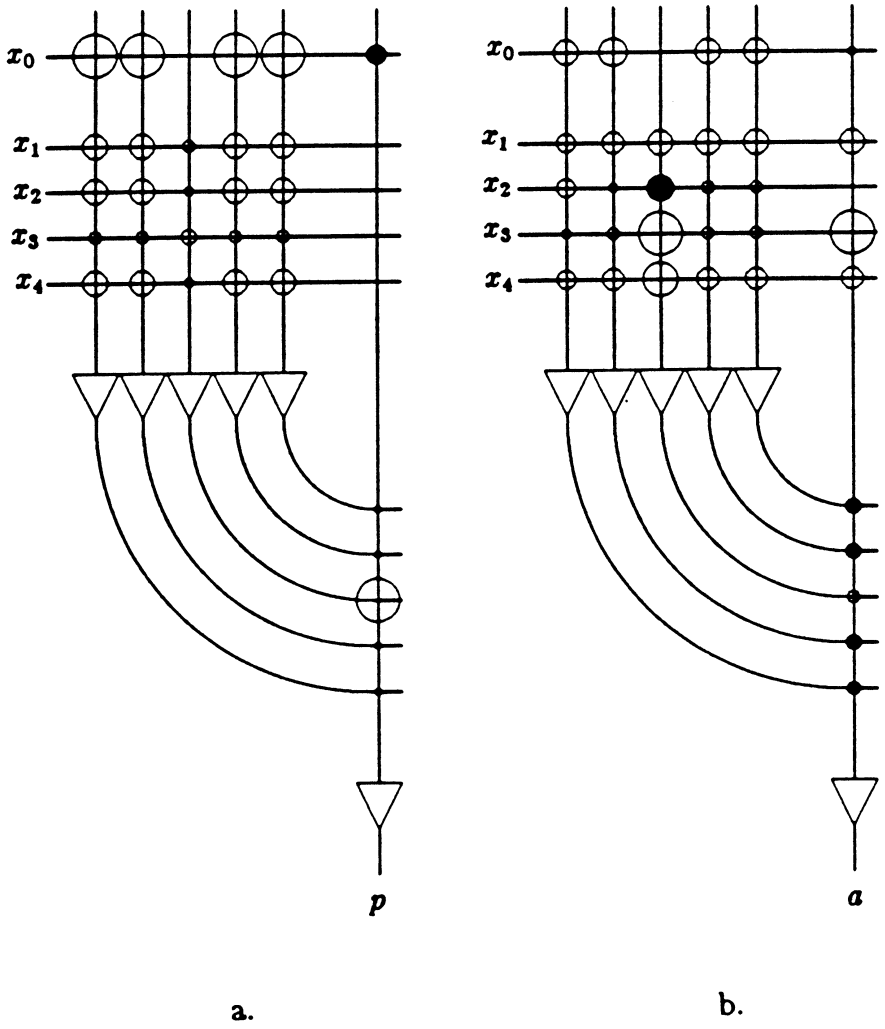
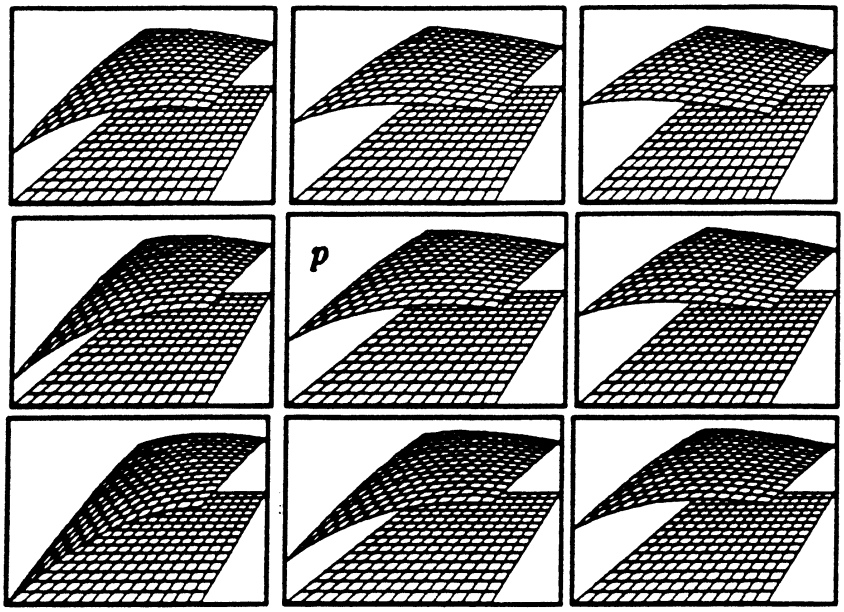
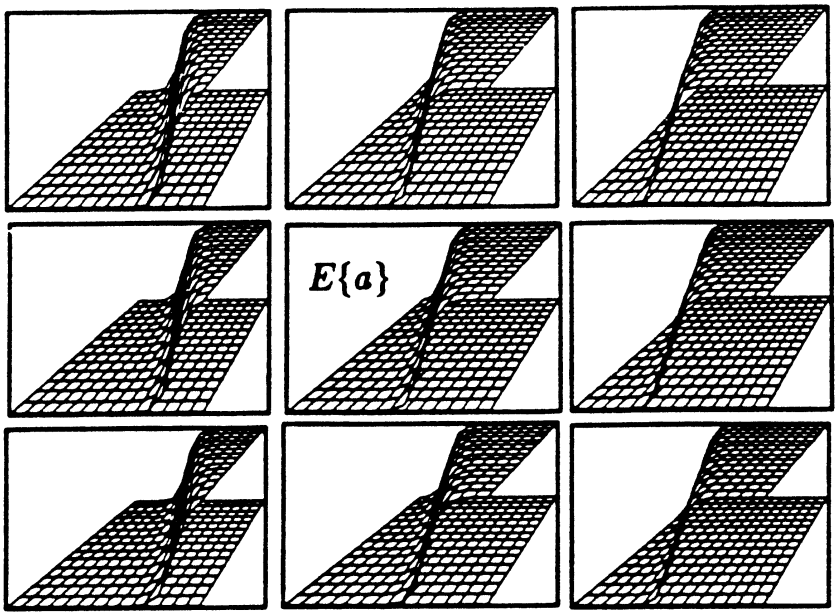


Figure 6.7: Weights Learned by Two-Layer Network



a.



b.

Figure 6.8: Functions Learned by Two-Layer Networks

New Features

Now we continue with the analysis of the hidden units. Unit 3 of both networks acquired significant effects on their output units. The functions implemented by their weights can be visualized by graphing them as functions of θ and $\dot{\theta}$ for different values of x and $\dot{x}$, as done for the functions implemented by an entire network. Figure 6.9a shows these graphs for Unit 3 of the evaluation network and Figure 6.9b shows the graphs for Unit 3 of the action network. The outputs of these units varies from 0 to 1. Very similar functions were learned by the two units. They both produce a fairly constant value of 1 for most states, with lower values approaching 0 when θ and $\dot{\theta}$ become more negative. However, the contribution of Unit 3 of the action network is very small—its output weight is small in comparison to the larger weights on the output unit, Unit 6. This is not surprising, since the desired mapping from state to action can be implemented with a single unit. In fact, setting Unit 3's output weight to 0 and adding its magnitude to Unit 6's constant-input weight causes little change in the state-to-action mapping.

To test the significance of the new feature learned by Unit 3 of the action network, further experiments were run with a one-layer action network and the two-layer evaluation network. The one-layer action network did learn the desired function, but it learned it slower than did the two-layer action network. Perhaps the feature learned by Unit 3 facilitated the learning of a good action function, and with additional experience the output unit developed the appropriate weights for its state-variable inputs. This must be verified by observing the evolution of the action function both as a function of the state variables and as a function of the hidden units' outputs.

The role of the evaluation network's Unit 3 is much more important. The hill-shaped evaluation function cannot be implemented without the hidden units, shown by the results of the single-layer experiments. Through its positively-weighted connection to the output unit, Unit 3 generates the positive gradient in the evaluation surface as we move from negative $\theta, \dot{\theta}$ to $\theta, \dot{\theta} = 0$. At this point, the gradient in the response of Unit 3 effectively becomes zero, and the output unit's negative weights on its state-variable inputs provide the negative gradient as we continue to move in the positive $\theta, \dot{\theta}$ direction.

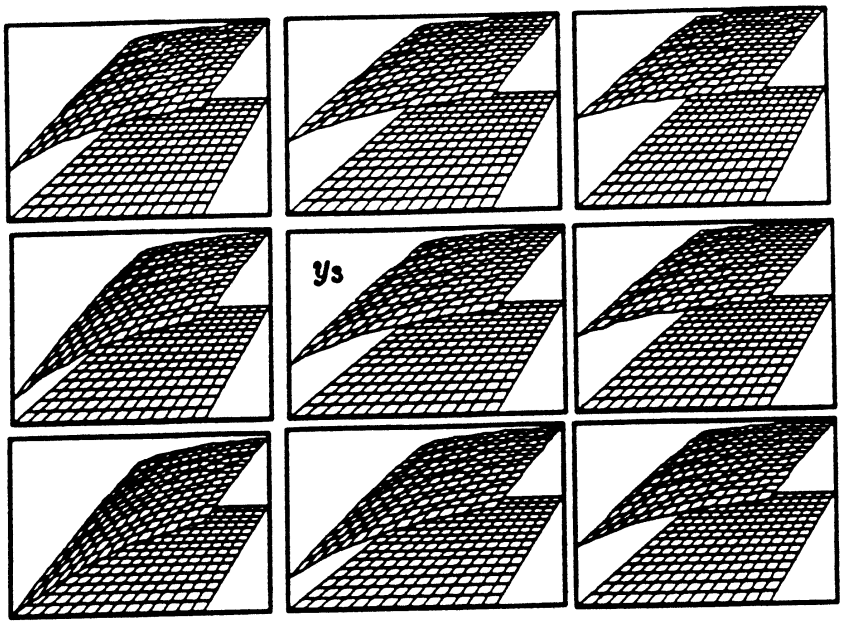
6.5 Transfer to Similar Tasks

Selfridge, Sutton, and Barto (1985) investigated the ability of a pole-balancing system to transfer learning between tasks. They applied the one-layer system designed by Barto, Sutton, and Anderson (1983) to a number of balancing tasks that differ in the values of the mass or length parameters, and discovered that training to one task facilitated training in subsequent, related tasks. Transfer was successful because for many states the related tasks required the same action or the same evaluation.

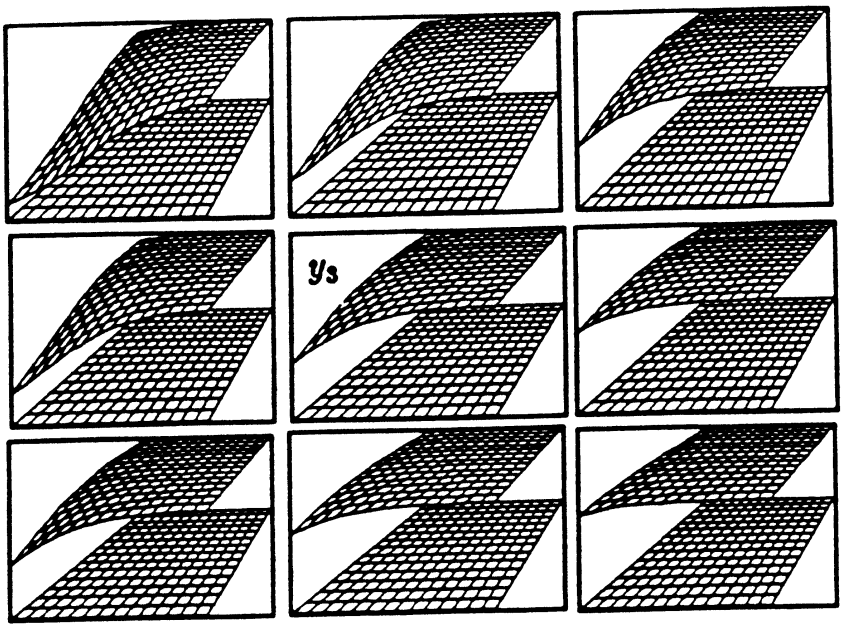
The two-layer learning system explored here would result in transfer of learning for the same reason. In addition, transfer would arise from the new features learned by the system. New features could decrease the time needed to learn a subsequent task, even if the task required the use of a novel set of actions. For example, an electric motor might be used to exert a force on the cart for one task, and a pneumatic piston might be used in a second task. Transfer of learning would not be caused by the use of the same action functions, but by the presence of features that are learned through experience with the first task. In some sense, a higher-level of knowledge about the domain would be learned—knowledge that is useful for a wider range of tasks, with different action sets and even with different goals. We didn't, alas, have time to show this.

6.6 Conclusion

It is immediately apparent from the learning curves of Figures 6.3 and 6.6 that the two-layer learning system far outperformed the one-layer learning system. New features are required for the formation of a good state-evaluation function and, although not required for the learning of a good action function, new features facilitated the action function's formation.



a.



b.

Figure 6.9: New Features Learned by Two-Layer Networks

Thus, the synthesis of the error back-propagation scheme of Rumelhart, Hinton, and Williams (1986) and the reinforcement-learning techniques (Barto, Sutton, and Anderson, 1983; Sutton, 1984) produced a learning algorithm for a connectionist network that successfully deals with delayed reinforcements and the initial lack of an adequate representation. The algorithm resulted in a controller that balanced the pole for 9 minutes (simulated time—28,000 steps at 0.02 seconds per step) and probably would have balanced it longer if the experiments had been run for a greater number of steps.

Comparison with the single-layer system of Barto, et al. (1983) is made difficult by the differences in how the experiments were conducted. The difference with the greatest effect is that Barto, et al. started the cart-pole system in the same state, $(x, \dot{x}, \theta, \dot{\theta}) = (0, 0, 0, 0)$, following every failure, whereas here the start state was selected randomly. This is one explanation for the lack of steady improvement followed by the large jump in the performance curve of Figure 6.6. Average performance is kept low by start states that are very close to failure states. Disregarding this difference, comparisons show that the system of Barto, et al. achieved a higher average trial length after 500,000 steps: their system resulted in approximately 80,000 steps per trial, while the experiments here resulted in approximately 30,000 steps per trial. This difference reflects the fact that the two-layer connectionist system learned good solutions later in the runs than did Barto, et al.'s, system. We conclude that a considerable number of steps are required for the hidden units to learn the necessary features—it is not until good features are learned that a useful evaluation function can be formed, and until the evaluation function is learned the action network cannot progress past a low level of performance.

The heuristic combination of reinforcement-learning and error-correction methods developed here is just an initial attempt at the formulation of algorithms for reinforcement learning in multilayer systems. The current algorithm did learn solutions to the pole-balancing task, but much experience—an average of 10,000 failures—was required before the pole could be balanced. Ways of accelerating the learning of new features would be fruitful.

The pole-balancing task is described in detail so others can directly compare the results of other learning techniques to the results reported here. More complicated versions of the pole-balancing task are easily generated, to create increasingly difficult tests for learning algorithms. Widening the bounds on the pole's angle introduces nonlinearities in the desired action function. A full 360-degree pole is a an interesting case—the controller must learn to switch strategies as the pole passes through the horizontal plane. The two-dimensional world of the cart-pole can be expanded to three dimensions. The parameters of the cart-pole can be varied over time, simulating disturbances arising from wind, wear between the cart and the track or on the pole's pivot, loss of pole mass, etc. The task can also be complicated by adding more poles to be balanced, or by using a flexible pole (Schaefer and Cannon, 1966). Additional inputs to the learning system that encode additional factors, such as the number of poles or the wind speed, would result in functions that are dependent on these variables, and the system would react appropriately when they abruptly change. One further complication is introduced when an explicit representation of the goal is presented to the system. For example, a system could learn to balance the pole upright, to point the pole downward, dampening all motion, or to spin the pole as quickly as possible, choosing its actions based on which of these is the current goal.

A learning system that can solve this pole-balancing task has potential application in a number of related control domains. The dynamics of the inverted pendulum are closely related to those of other systems for which balance must be maintained. Raibert (1986) reviews research on walking machines, and mentions its relationship to the work of Cannon and his colleagues on the design of controllers for inverted pendulums (Higdon and Cannon, 1963; Schaefer and Cannon, 1966; Cannon, 1967). Another related control problem is the stabilization of rockets through the control of the direction of thrust.

Chapter 7

Learning the Solution to a Puzzle

Connectionist representations present a natural approach to the learning of solutions to numerical control tasks, as illustrated in Chapter VI. In this chapter, we show that by casting the solution of a problem-solving task as a mapping between numerical representations of problem states and actions, the connectionist system used in Chapter VI for a numerical control task can be applied with little modification to a problem-solving task whose solution is typically represented symbolically (as production rules, for example). As before, the connectionist system consists of two networks: an evaluation network which learns an evaluation function, and an action network which learns heuristics that guide the search for good actions. Comparisons of methodology are made with Langley's (1985) adaptive production system, called SAGE, which learns heuristics that improve the performance of an initial weak search strategy. To facilitate this comparison, we selected the three-disk Tower of Hanoi puzzle for our experiments, one of the puzzles that Langley used to demonstrate SAGE.

7.1 The Tower of Hanoi Puzzle

The Tower of Hanoi puzzle is popular for research in problem-solving because the number of states is small, but the puzzle is still difficult to solve. Human strategies for solving the Tower of Hanoi have been analyzed (Luger, 1976) and modeled (Anzai and Simon, 1979). Amarel (1981) used the Tower of Hanoi puzzle as a vehicle for studying shifts of representations to forms of increasing efficiency for the discovery of a problem's solution.

The state of the Tower of Hanoi puzzle can be represented in a number of ways. A common representation is one used by Nilsson (1971), for which the pegs are numbered 1, 2, and 3, and the disks are labeled A, B, and C, where Disk A is the smallest disk and C is the largest. A particular state is represented by the peg numbers where each disk resides, listed for disk C, then disk B and disk A. As pictured in Figure 7.1, the initial state of the puzzle is (111), and the objective is to achieve state (333) by applying a sequence of actions. The only legal actions are movements of the top-most disk from one peg to another, with the restriction that a disk may never be placed upon a smaller disk. An action may be represented as a source peg and destination peg, so the transformation of state (111) to state (112) is performed by the action of moving the top-most (smallest) disk from peg 1 to peg 2, represented by Action 1-2. For the three-peg puzzle, only six actions exist: 1-2, 1-3, 2-1, 2-3, 3-1, and 3-2.

The states of the puzzle plus the transitions between the states corresponding to the legal actions form the puzzle's state transition graph, as shown in Figure 7.2. To evaluate a human's or machine's ability to improve its search strategies on the Tower of Hanoi puzzle, we measure the number of actions in the solution path—a route through the state transition graph from the initial to the goal state—the minimum length path being the objective. For the three-disk puzzle, the minimum-length solution path has seven actions and is the straight path down the right side of the state transition graph. Finding the shortest solution path is confounded by the large number

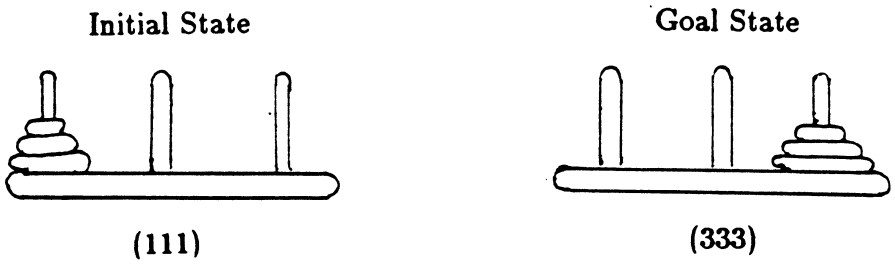


Figure 7.1: Initial and Goal States of the Tower of Hanoi Puzzle

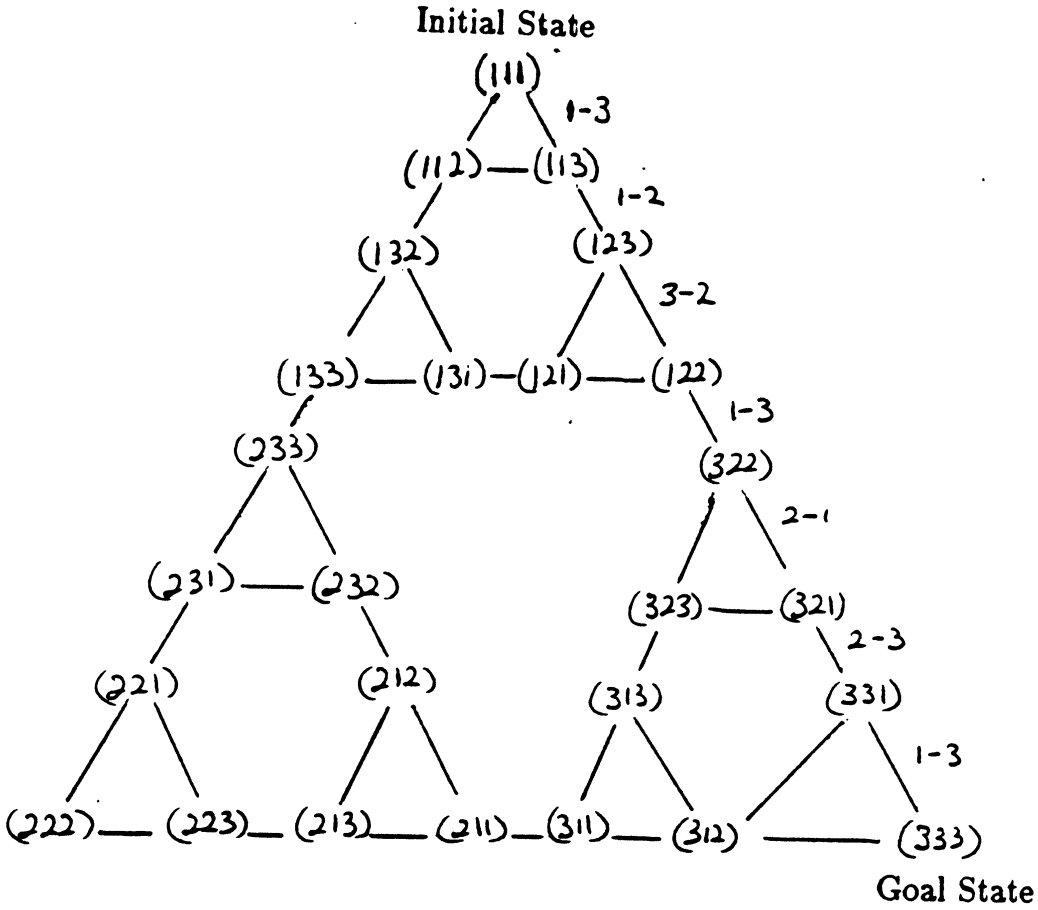


Figure 7.2: State Transition Graph for Three-Disk Tower of Hanoi Puzzle
(Adapted from Nilsson, 1971)

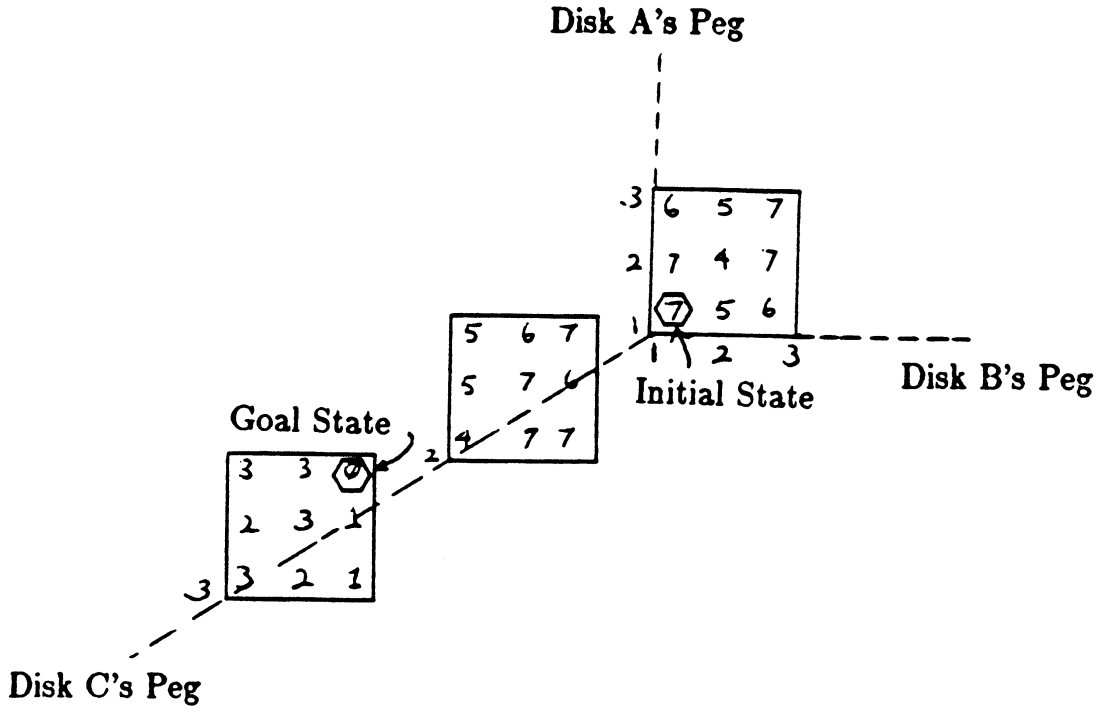


Figure 7.3: 3-Dimensional State Space and Desired Evaluations

of possible solution paths and by the presence of many loops, or cycles, in the state transition graph.

The Tower of Hanoi puzzle is a good test of the multilayer connectionist system developed in Chapter V, particularly of its ability to generate new features, for the following reason. A useful evaluation function must map states to values that indicate the states' closeness to the goal node. For the state representation described above, this mapping can be visualized by considering the peg numbers to be dimensions of a three-dimensional state space, as shown in Figure 7.3. Each point corresponding to a state is annotated by the number of actions required to reach the goal state. A linear threshold unit, or other unit based on a linear weighted sum of its inputs, cannot make the necessary discriminations between states to form a good evaluation function. New features must be formed by a layer of hidden units that carve up the state space into a representation for which the linear output unit can learn a good evaluation function. For the experiments described in this chapter the representation was simplified somewhat, as described later, to reduce the time required to learn the solution. The fact that new features are still required is shown by the inability of a one-layer system to learn the minimal solution path.

The formation of useful search heuristics for the Tower of Hanoi is less complicated. A small set of rather simple heuristics can constrain the search to exactly the correct actions (Anzai and Simon, 1979; Langley, 1985). For example, many alternatives are removed by a rule stating that it is undesirable to apply the inverse of an action. The action network used to learn search heuristics in the following experiments is single-layered, and did successfully learn the minimal solution path. The representation provided to the network is sufficient—new features are not required. The simplicity of the search heuristics does not lessen the need for a good evaluation function; credit must be assigned correctly in order for the heuristics to be learned.

7.2 Experiments

As done for the pole-balancing experiments, the performance of a one-layer connectionist system is compared to the performance of a two-layer system to demonstrate the development of useful features. The learning algorithms and the networks are very similar to those used for the pole-balancing task, with small modifications to the network structure and the algorithms. Before describing these modifications, the state and action representations are discussed. Then the results of experiments with the one and two-layer networks are presented.

7.2.1 Representation of States and Actions

As mentioned above, the state representation consisting of the three peg-numbers corresponding to each disk results in a very complex mapping from states to evaluations. Although in principle the connectionist systems used here should be able to learn this mapping, we wished to simplify the task somewhat to reduce the simulation time required for the experiments.

The state representation used in the following experiments is composed of nine binary digits. The first three digits encode Disk C's peg number, the second encode Disk B's peg, and the third set of three digits encode Disk A's peg. Peg 1 is encoded as 100, Peg 2 as 010, and Peg 3 as 001. For example, state (111) is represented as (100 100 100), and state (123) is represented as (100 010 001).

The output of the action network represents an action by a six-component, standard unit basis vector, where the components correspond to actions 1-2, 1-3, 2-1, 2-3, 3-1, and 3-2, respectively; Action 1-2 is encoded as (100000), Action 1-3 is (010000), and so on.

Both the evaluation network and the action network receive the representation of the state. This completely defines the input to the evaluation network, but additional terms are presented to the action network. We wished to investigate the ability of the connectionist network to learn search heuristics similar to the rules developed by Langley's SAGE system. As mentioned above, one such rule is to never apply the inverse of the previous action. In order to learn such an association between the previous action and the current action, the previous action must be provided as input to the action network. Another rule learned by SAGE is to not apply an action that returns the puzzle to a state that was visited two steps ago. This avoids the three-step loops around the smallest triangles in the state transition graph (Figure 7.2). Rather than providing past states as input, we chose to present the action taken two-steps ago, in addition to the previous action. The previous two actions along with the current state provide enough information to identify the state visited two steps ago, although our results suggest that hidden units are needed to overcome the linearity of the output unit, perhaps by forming a conjunction of the previous two actions. This possibility was not investigated.

7.2.2 Credit Assignment

The evaluation network learns an evaluation function, but in its initial state the network implements a meaningless function that evaluates all states approximately equally since its weights are initialized to small random values. With experience it converges on a function that predicts the occurrence of reinforcements.

The most significant reinforcement occurs whenever the goal state is entered. A reinforcement value, labeled $r[t]$, of 1 is presented for the time step at which the goal state (333) is entered. Recall that for the pole-balancing task, the goal is to avoid certain states for as long as possible, and $r[t]$ was set to -1 upon entering those states. The Tower of Hanoi task could be solved (by a two-layer network) with only this final reinforcement, but two additional reinforcements are provided for the following reasons.

If the action probabilities converge too quickly, due to a large value for the parameter ρ, a solution path of longer than minimum length will probably be learned. For example, say the learned solution path is of length eight, one step longer than the minimum number, due to the incorrect Action 1-2 being taken from the starting state (111). If this action is always chosen over the correct Action 1-3, then an evaluation function tailored to this particular solution path will be learned. To avoid this, a second reinforcement signal is presented having a constant value of -0.1

for all non-goal states. In this way, a shorter solution path results in a higher total reinforcement than does a longer solution path. This parallels the role of Langley's heuristic for judging shorter paths between two states as more desirable.

The third reinforcement is a value of -1 presented whenever a two-step loop occurs, i.e., when the current action reverses the effect of the previous action. The random search initially followed by the action network results in many two-step loops (and longer loops), thus many steps elapse before the goal is discovered. The large negative reinforcement results in a significant decrease in the probability of selecting the action that is the inverse of the previous action. As shown later, this must be learned for each action—the concept of inverse actions is not known to the system, so generalizing across actions is not possible. This reinforcement is not presented to the evaluation network, enabling a large negative reinforcement to be used without decreasing the evaluation for the corresponding state. The large negative reinforcement is not meant to indicate that a state is bad, only that the action was bad. The selection of the inverse action should be discouraged, but not necessarily the visitation of the state.

This third type of reinforcement is not necessary for the system to learn the puzzle's solution. It was included for two reasons. Firstly, it does significantly reduce the number of search steps during the early stages of learning. Secondly, it demonstrates how domain knowledge, such as the undesirability of one-step loops, can be added by altering the reinforcement function.

7.2.3 Interaction between Learning System and Puzzle

The input to the evaluation network is composed of the nine binary digits that encode the state of the puzzle, plus a constant component with a value of 0.5. Specifically, the input terms, $x_i[t]$, for the state at time t are given by:

$$
\begin{aligned}
x_0[t] &= 0.5, \\
x_1[t] &= \begin{cases} 1, & \text{if Disk C is on Peg 1 at time } t; \\ 0, & \text{otherwise,} \end{cases} \\
x_2[t] &= \begin{cases} 1, & \text{if Disk C is on Peg 2 at time } t; \\ 0, & \text{otherwise,} \end{cases} \\
x_3[t] &= \begin{cases} 1, & \text{if Disk C is on Peg 3 at time } t; \\ 0, & \text{otherwise,} \end{cases}
\end{aligned}
$$

and similarly for $x_4[t]$, $x_5[t]$, $x_6[t]$ and Disk B, and for $x_7[t]$, $x_8[t]$, $x_9[t]$ and Disk A. After the goal state is reached and at the start of every run, the state is set to (111), so the input becomes (100 100 100) disregarding the constant input.

The input to the action network consists of these terms and, in addition, the previous two actions. The action at time t is encoded by six binary components, $a_1[t], a_2[t], \ldots, a_6[t]$,[1] so the additional input terms are defined as:

$$
\begin{aligned}
x_{10}[t] &= a_6[t-2], \\
&\vdots \\
x_{15}[t] &= a_1[t-2],
\end{aligned}
$$

and

$$
\begin{aligned}
x_{16}[t] &= a_6[t-1], \\
&\vdots \\
x_{21}[t] &= a_1[t-1].
\end{aligned}
$$

[1]The indices of the action variables have been renumbered to start from one. In Chapter V they were indexed by the output unit indices $m_h, \ldots, m$.

t	State	Action	State $x_1...x_9$	Delayed Actions $x_{10}...x_{15}$	Delayed Actions $x_{16}...x_{21}$	Action $a_1...a_6$	Evaluation $p[t]$	Reinforcement $r[t]$
1	111	1-3	100100100	000000	000000	010000	−0.32	−0.1
2	113	1-2	100100001	000000	010000	100000	−0.17	−0.1
3	123	3-2	100010001	010000	100000	000001	−0.14	−0.1
4	122	1-3	100010010	100000	000001	010000	0.10	−0.1
5	322	2-1	001010010	000001	010000	001000	0.35	−0.1
6	321	2-3	001010100	010000	001000	000100	0.51	−0.1
7	331	1-3	001001100	001000	000100	010000	0.75	1.0
8	333							

Table 7.1: Interaction of Learning System and Tower of Hanoi Puzzle

The possible values of reinforcement are defined as follows. The value of $r[t]$ includes just the first two kinds of reinforcement, while the one-step loop penalty is given by $r_{\text{loop}}[t]$ to distinguish between the reinforcements that are and are not presented to the evaluation network:

$$r[t] = \begin{cases} 1.0, & \text{if state at time } t \text{ is (333);} \\ -0.1, & \text{otherwise.} \end{cases}$$

$$r_{\text{loop}}[t] = \begin{cases} -1.0, & \text{if state at } t \text{ equals state at } t-2; \\ 0.0, & \text{otherwise.} \end{cases}$$

An example of the interaction between the learning system and the Tower of Hanoi puzzle is shown in Table 7.1. The sequence is taken from a successful experiment, after the networks have learned good evaluation and action functions. The correct action is taken on each step, and the evaluation, p, increases as fewer steps remain to reach the goal state.

7.2.4 Results of One-Layer Experiments

The only modification to the learning algorithms used for the pole-balancing task involves the equation for updating the weights of the action network. The reinforcement signal for one-step loops, r_{loop}, is added as follows:

$$w_{i,j}[t] = w_{i,j}[t-1] + \rho\,(r_{\text{loop}}[t] + \hat{r}[t])\,\Big(a_j[t-1] - E\{a_j[t-1]|w;x\}\Big)\,x_i[t-1].$$

As in the pole-balancing experiments, two performance measures were used to select the best values for the parameters of the weight-update equations. A measure of cumulative performance throughout a run is provided by the number of trials (achievements of the goal state) averaged over all runs for a given set of parameter values. The second performance measure is the average over all runs of the number of steps in the last trial, or the preceding trial, whichever is smaller.

The final performance level averaged over 5 runs of 50,000 steps each was used to select the best of approximately 20 sets of parameter values, differing in ρ and β, leaving $\gamma = 0.9$. The best values were:

$$\begin{aligned} \beta &= 0.100, \\ \rho &= 0.01, \\ \gamma &= 0.9. \end{aligned}$$

These values were used in a longer experiment of 10 runs of 100,000 steps each, resulting in the number of trials and last trial lengths shown in Figure 7.2. The average number of trials is $3,145$. From the lengths of the last trial for each run we see that the minimal solution path was not learned in any run—all trials are longer than seven steps. Run 7 resulted in a last trial of length nine, but it wasn't determined whether the path taken on the final trial would be reliably followed

Run	Trials	Last Trial
1	2,911	28
2	2,877	23
3	2,884	19
4	2,893	65
5	2,686	651
6	2,928	25
7	4,481	9
8	2,902	25
9	3,951	38
10	2,940	41

Table 7.2: Results of One-Layer System

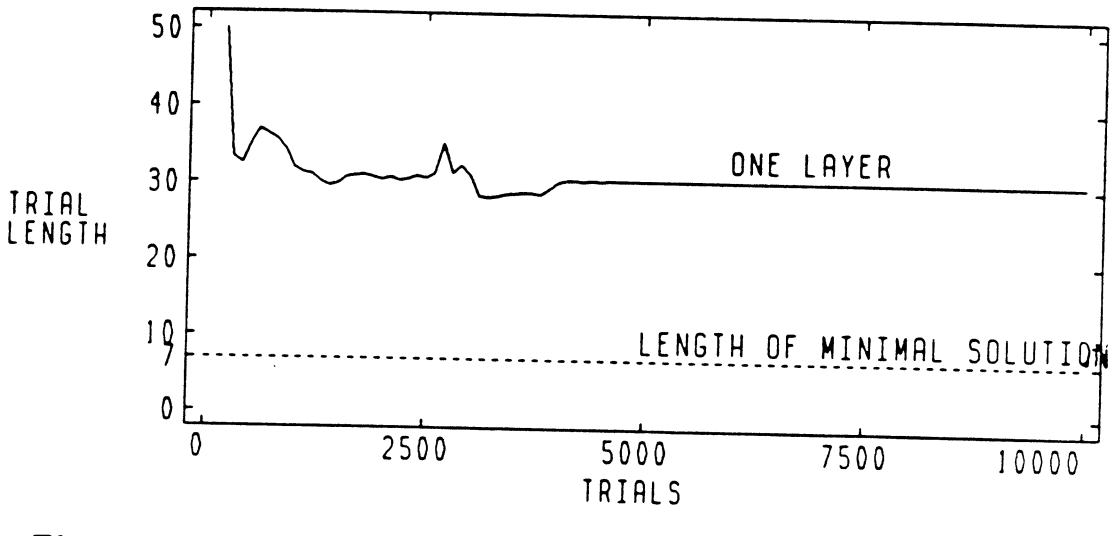


Figure 7.4: Length of Solution Path versus Trials for One-Layer System

for subsequent trials. The low number of total trials for Run 7 indicates that paths longer than nine steps are likely.

The trial length versus the number of trials is plotted in Figure 7.4, showing how the length of the solution path varies with experience. The horizontal dotted line in the figure is at a trial length of seven, the length of the minimal solution path. The values plotted are averages over the 10 runs and over bins of 100 trials. The length of the solution path decreases quickly from an average of 50 steps to approximately 30 steps, but performance is never much better than 30 steps per solution. A non-learning strategy of random action selection was found to result in an average of 140 steps per solution path, so the one-layer system significantly improves the initial random search strategy. Note that all runs were terminated before 5,000 trials elapsed—the learning curve was extended as was done for the pole-balancing experiments. The curve might have continued to decrease slightly if the one-layer experiments had been run longer.

The weights learned by the end of Run 7 are shown in Figure 7.5. Actual values are provided in Appendix C. The evaluation network has acquired only three weights of significant magnitude, and they are all associated with the encoding of Disk C's peg. The weights' signs result in high evaluations for states with Disk C on Peg 3, the goal peg, and low evaluations for other states. The weights of the action network are more difficult to interpret. Let us postpone the discussion of these weights until the results of the subsequent experiments with a two-layer evaluation network are presented.

We can visualize the learned evaluation function by viewing the state transition graph and

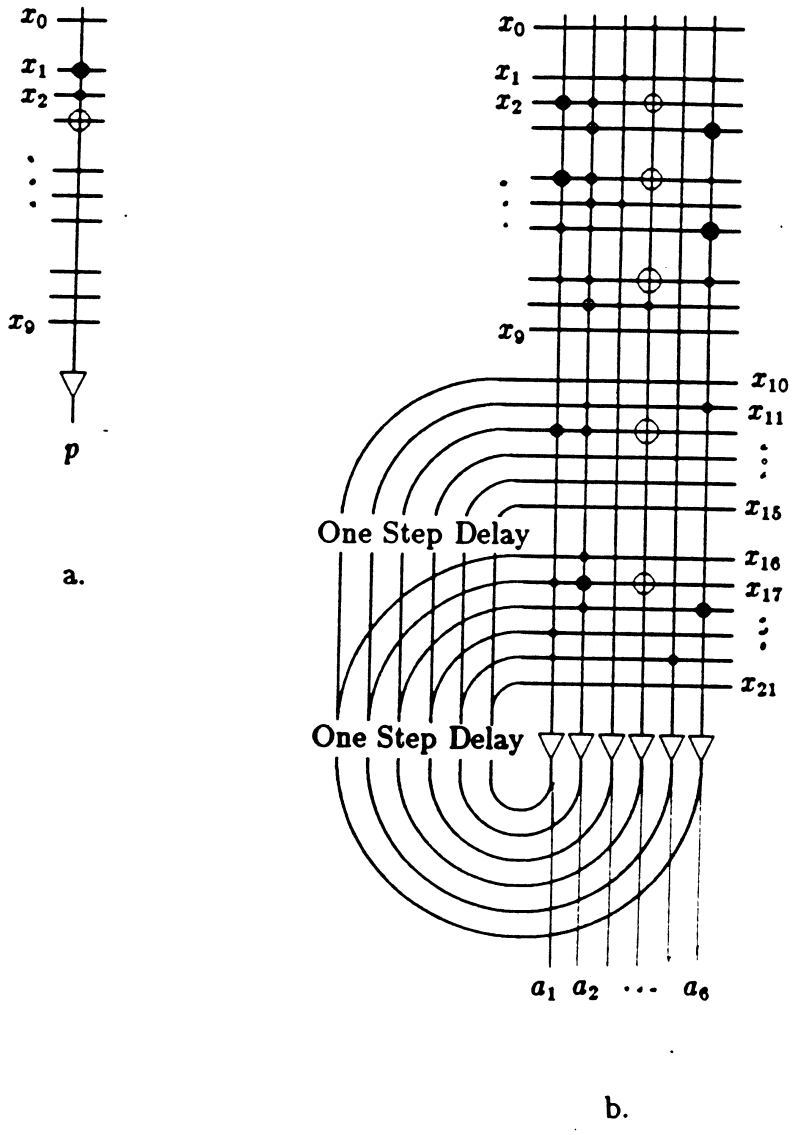


Figure 7.5: Weights Learned by One-Layer Network

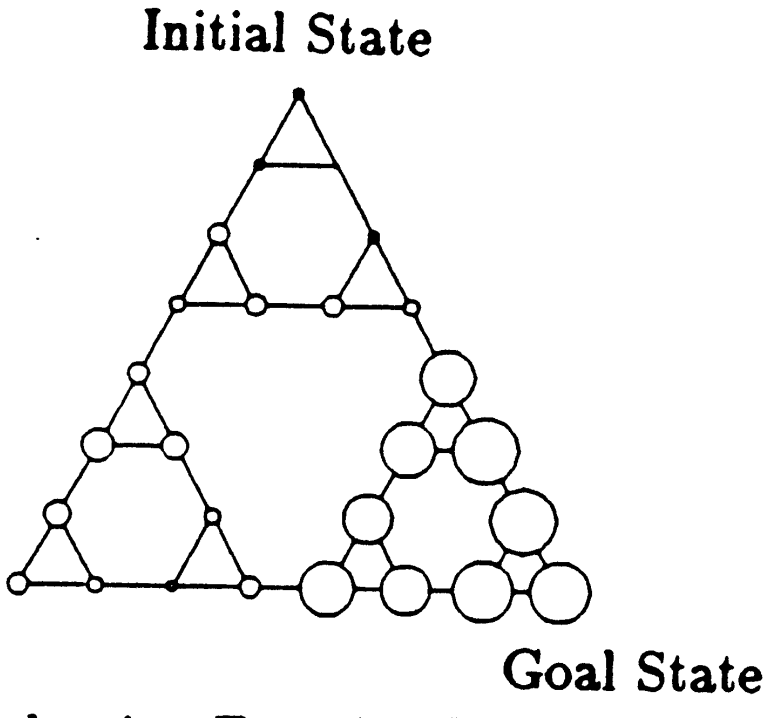


Figure 7.6: Evaluation Function Learned by One-Layer Network

at each state (i.e., node in the graph) drawing a circle with radius proportional to the state's evaluation. The evaluation function learned in Run 7 is pictured in this manner in Figure 7.6. As determined from the signs of the weights, the evaluation function indeed produces high values for states for which Disk C is on Peg 3, which are the states in the large, lower right triangle of the state transition graph. There is very little additional information provided by this evaluation function. We can describe this function as a credit-assignment heuristic, viz., states with Disk C on Peg 3 are desirable.

7.2.5 Results of Two-Layer Experiments

Our two-layer experiments involved a two-layer evaluation network with 10 hidden units, but with the same one-layer action network as above. We suspected that with the delayed actions as input terms, the one-layer action network could find weight values that result in following the minimal solution path. This is verified by the results of the experiments.

Approximately 20 sets of parameter values were tested by performing 5 runs of 50,000 steps each. The values resulting in the best performance are:

$$\begin{aligned} \beta &= 0.1 \\ \beta_h &= 2.0 \\ \beta_m &= 0.9 \\ \rho &= 0.02 \\ \gamma &= 0.9 \end{aligned}$$

Momentum was discovered to facilitate learning in this case, though interestingly it retarded learning for the pole-balancing task. Applying these values in 10 runs of 100,000 steps produced the results in Table 7.3. For all but one run the length of the last trial was seven steps, equal to

Run	Trials	Last Trial
1	11,809	7
2	11,418	22
3	11,584	7
4	11,559	7
5	11,967	7
6	12,093	7
7	11,856	7
8	11,636	7
9	12,432	7
10	12,041	7

Table 7.3: Results of Two-Layer System

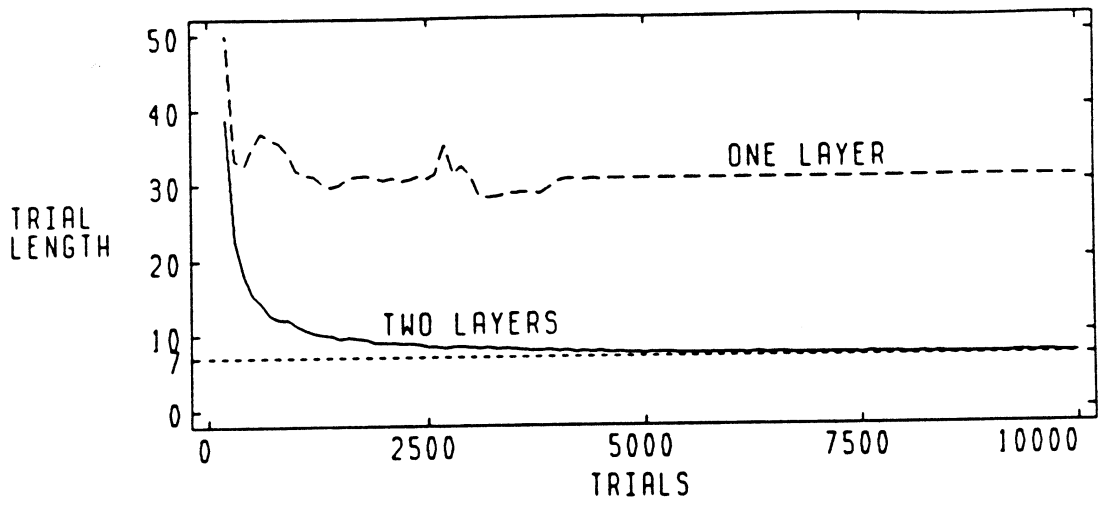


Figure 7.7: Length of Solution Path versus Trials for Two-Layer System

the length of the minimal solution path. The average number of trials is 11,839, roughly 10 steps per trial averaged over the 100,000 steps. So in 9 out of 10 runs the minimal solution path was learned, and judging from the number of trials in Run 2, the minimal solution path was probably reliably followed in that run, also. There is always a nonzero probability of trying an alternative path, which could explain the last trial of Run 2.

The learning curve of Figure 7.7 shows that the two-layer system quickly learned solution paths averaging about 15 steps in length, and gradually reduced this to the minimum of seven steps. The learning curve for the one-layer system is superimposed on this graph to highlight the performance increase resulting from the hidden units in the evaluation network.

The weights learned during Run 1 are displayed in Figure 7.8. (See Appendix C for actual values.) First we focus on the weights of the evaluation network. The hidden-unit weights are more varied than they were for the pole-balancing task. Most of the units appear to have acquired useful new features. Units 1, 2, 5, 6, and 9 have weights of large magnitude, though they are by no means the only units of significance. As is usually the case, it is difficult to comprehend what role the units play by studying individual weight values. However, by encoding their output values by the size of circles on the state transition graph, as done earlier for the evaluation function itself, we can learn exactly what the new features are and can gain some intuitions about their contributions to the overall evaluation function. First, we analyze the evaluation function that was learned in Run 1.

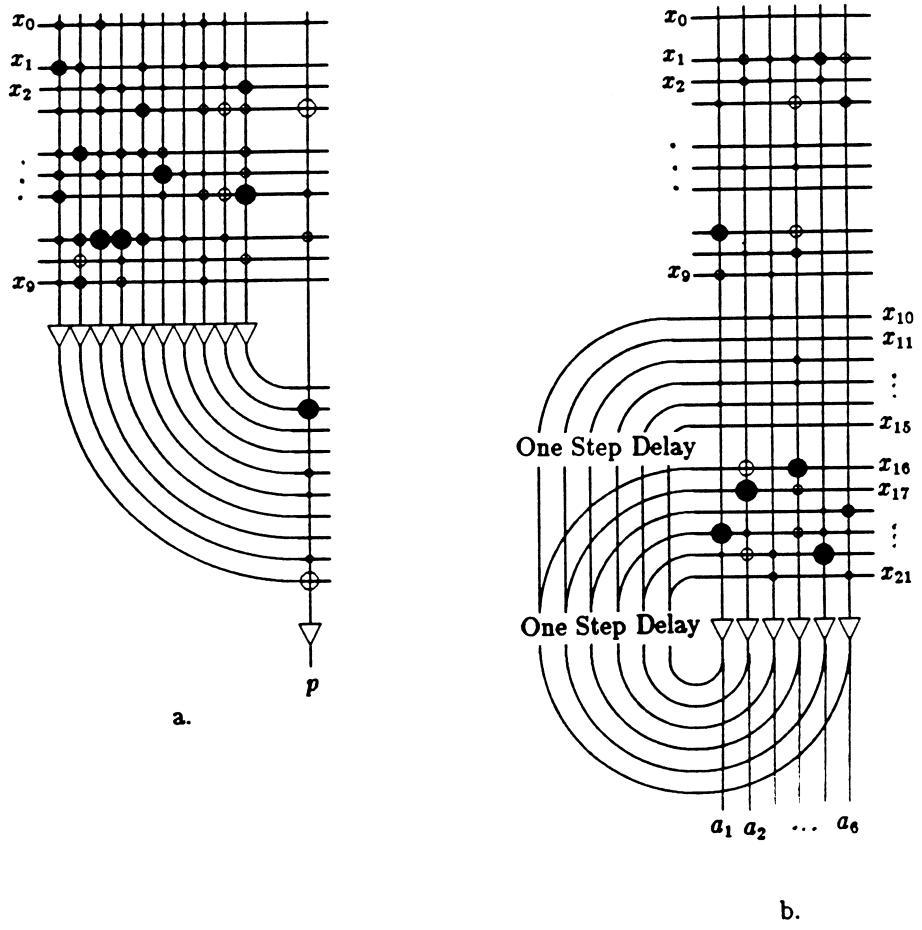


Figure 7.8: Weights Learned by Two-Layer Network

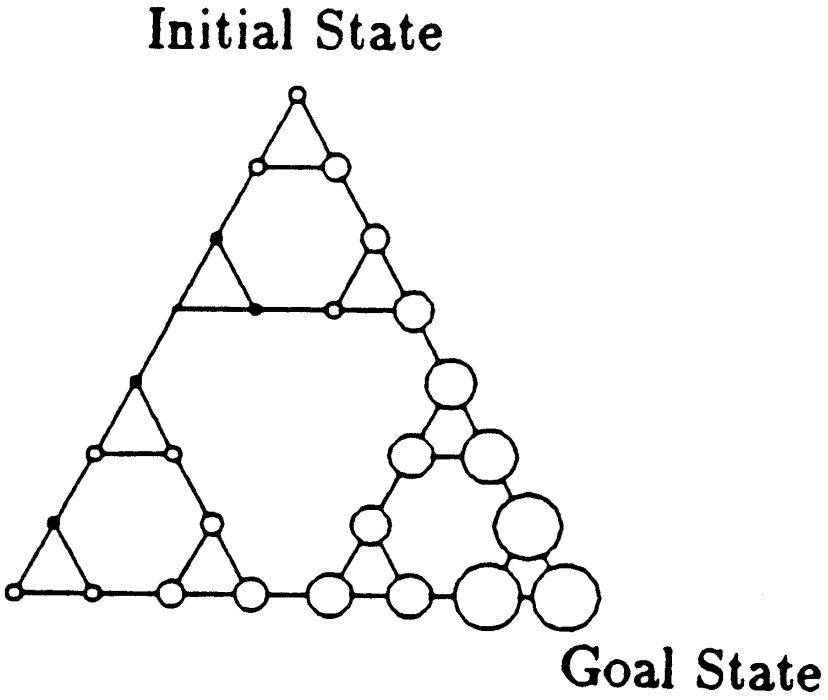


Figure 7.9: Evaluation Function Learned by Two-Layer Network

7.2.6 The Evaluation Function and New Features

The value of the evaluation function learned in Run 1 is represented in Figure 7.9. In comparing two states, the state with the larger circle would be evaluated as being more desirable. Notice that a consistent progression from small circles to larger circles results as one moves from any state toward the goal state by the shortest route, thus this evaluation function is extremely informative. Any search strategy, in addition to the probabilistic method used to generate actions, would benefit from this evaluation function. The sample of the interaction between the learning system and the Tower of Hanoi puzzle shown in Table 7.1 is actually a trial from the end of Run 1, and one column of that table is the output value of the evaluation network. The value steadily changes from a negative value to increasing positive values as the goal state is approached.

Now let us see how this evaluation function is constructed. Figure 7.10 shows the output functions for the 10 hidden units, i.e., the features acquired during learning. The radii of the circles for a feature are calculated by scaling the 27 output values for the corresponding hidden unit to be between 0 and a maximum radius. So circles of extreme size do not necessarily indicate that the output is 0 or 1, but only that the output is a minimum or maximum of the values for that unit. We will not attempt to explain every feature, but will consider the contributions of several. We refer to a feature by the corresponding unit number, such as Feature 1 for the function learned by Unit 1.

Feature 1 has a positive effect on the evaluation. Figure 7.10 shows that Feature 1 roughly represents three states near the bottom of the graph just outside of the lower right triangular region where Disk C is on the goal peg. Feature 1 boosts the evaluation of these states, thus directing a search from states in the lower left part of the graph toward the state through which the search must pass to get Disk 3 onto the goal peg. This part of the graph is a "bottleneck", and similar bottlenecks exist at the other two junctions of the three largest triangles. The values of the evaluation function are critical near these bottlenecks—the choice of an incorrect action can

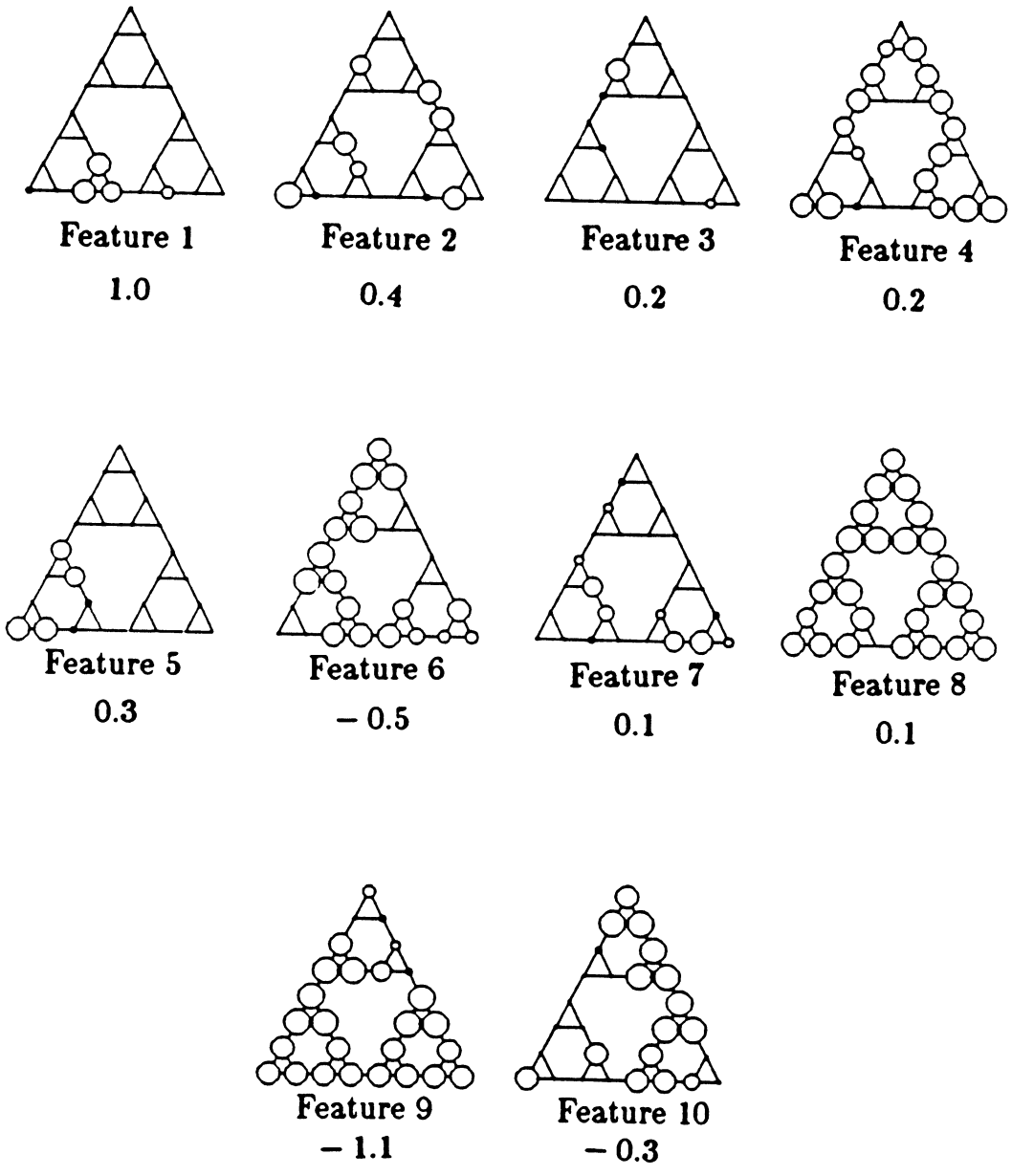


Figure 7.10: New Features Learned by Two-Layer Evaluation Network

result in many additional moves to return to the bottleneck to try a different action. Features 1 and 2 seem to be particularly helpful in evaluating the lower bottleneck and the bottleneck on the right, respectively.

Feature 9 has a very strong negative influence on the evaluation function. The value of Feature 9 is high for all states except the first four states on the minimal solution path, plus one nearby state. The evaluations of the first states in the solution path are raised in relation to the evaluations of the other states, thus Feature 9's role is to make the first few states of the minimal solution path more desirable than states next to the path.

Feature 10 also has a negative effect. Mainly the evaluations of states along and next to the minimal solution path are lowered by Feature 10, with the exception of the very last state before the goal state. It appears that this feature guarantees that the difference in state evaluations is positive as the last state is reached.

Other features also have important contributions. Perhaps a good way to understand their roles is to observe changes in the evaluation function as each feature is removed and then restored. From the small amount of analysis done here, it is clear that a variety of new features were developed for this task. The initial representation of the state is far from ideal when it comes to forming the evaluation function, but the combination of the error back-propagation algorithm with Sutton's AHC algorithm successfully learned sufficient new features for the state representation.

7.2.7 The Action Function

When the two-layer evaluation network was used, the single-layer action network was able to constrain the search to the minimal solution path. Figure 7.8b shows that this was accomplished mainly through the development of weight values for the previous-step's action and for the current state. The two-step delayed action did not acquire a significant effect on the selection of the current action.

The large negative weights on the previous action components stand out. Note that there is one large negative weight for each component. These weights have the same effect as did Langley's heuristic for preventing the application of the inverse of the previous action. By tracing the delayed output of each unit to the corresponding negative weight, we find that the negative weights are on the intersections of actions and their inverses. In other words, Action 1-2, or $a_1[t-1]$, has a negative connection to Action 2-1, or $a_3[t]$, Action 1-3 is negatively connected to 3-1, etc. A negative weight lowers the probability that the corresponding action will be generated, and these weights are of such high magnitudes that the probability of the previous-action's inverse is effectively zero.

There are other weights associated with the previous-action inputs that are positive-valued. Through these weights, the generation of an action on one step results in a high probability for a particular action on the next step, thus forming two-step *sequences*. For example, Action 3-2 will be followed by Action 1-3. Referring back to Figure 7.2 on page 118, this two-step sequence can change the puzzle from the third state on the minimal solution path to the fifth state. Other sequences exist for other two-step transitions along the minimal solution path, and for moving onto the minimal solution path.

The weights on the delayed action components are not sufficient in themselves for limiting actions to movement along the minimal solution path. The current state must at least play a role in selecting the very first action, and indeed it does. Consider the values of the input terms when in the start state (111). All of the delayed-output terms are 0, since this is the first step in the trial. All other input terms are 0, except for the first term of each of the three triples encoding the state. The first of these is connected positively to Unit 2 and negatively to the rest, except for Unit 6, whose action is not legal for the start state. The other two non-zero input terms have small or negative connections to units having legal actions, so Unit 2 will be the unit to respond to the start state. Unit 2 represents Action 1-3, the first action along the minimal solution path. Langley's system was not required to learn the correct action for the initial state, because both states (333) and (222) were goal states—two minimal solution paths exist, and both actions from the initial state (111) move along one of the paths.

7.3 Transfer of Learning

It is desirable for a learning system to be able to improve its performance on a single task, called *improvement*, and to also improve performance over a set of tasks, called *transfer* (distinction by Ohlsson, 1982). Langley (1985) lists the following four kinds of transfer between tasks:

1. Transfer to more complex versions of the task.
2. Transfer to different initial states or goal states.
3. Transfer to tasks of similar complexity with different state-space structures.
4. Transfer to tasks of little similarity, perhaps requiring some of the same actions (referred to as *learning by analogy*).

The ability of the connectionist learning system of this chapter to perform the first two kinds of transfer are discussed below.

Langley showed that the heuristics learned by SAGE for solving the three-disk Tower of Hanoi puzzle were directly applicable to the four-disk and the five-disk versions of this puzzle, solving these more complex puzzles with no additional search. The representation of the rules' conditions and actions made this possible: disk, peg, and action names are generalized to variables, therefore the rules can be applied to the new task having an additional disk, since its name can be bound to a variable as well as any previous disk name. In addition, the concept of an action's inverse in included in the system's representation, enabling a situation where an action is followed by its inverse to result in a rule discouraging the use of the inverse of *any* action.

The connectionist representation used here does not permit such generalizations, though learning is transferred to Tower of Hanoi puzzles having more disks. The state transition graph of the four-disk puzzle can be generated by duplicating the three-disk puzzle's graph three times, making an upper, lower left, and lower right major triangle to be connected in a structure identical to that for the three-disk graph, as shown in Figure 7.11. If the input representation of the networks is augmented by simply adding three components to encode the position of the new fourth disk, assumed to be the largest disk for this discussion, then the action function resulting in the minimal solution path learned for the three-disk puzzle is left intact, as highlighted in the figure. Due to the recursive nature of the Tower of Hanoi family of puzzles, the solution path appears three times. This solution path from the initial state (1111) actually moves the four-disk puzzle away from the goal state (3333). The action associated with the initial state is wrong, but there are associations that are appropriately transferred. The negative weights preventing the selection of an action that is the inverse of the previous action are still very helpful for the four-disk puzzle. Some of the two-step sequences might also be applicable. To learn the four-disk solution, the evaluation function must also be adjusted, since it is very tailored to the three-disk version. Therefore, the solution of the four-disk puzzle would require additional learning, though probably less than would be needed by a naive system that has no experience with the three-disk puzzle.

The second form of transfer concerns different initial and goal states. Langley's system was not capable of transferring to Tower of Hanoi puzzles with different initial and goal states, but he has shown on another task how the inclusion in the system of a representation of the goal can lead to strategies that are goal-dependent.

The action function learned by our system might generalize correctly to different initial states, particularly those close to the minimal solution path, but this was not tested. The evaluation function does generalize correctly to different initial states. As shown in Figure 7.9, the evaluations increase for states closer to goal, whether or not the states are on the minimal solution path if the new goal is not close to the original goal. Therefore, learning would be facilitated if the initial state were changed from its original position after the evaluation function had been learned.

As for different goal states, both action and evaluation networks have learned inappropriate functions. In fact, generalization to a puzzle with a different goal state would retard the learning of a new solution path. As Langley suggested, to learn evaluation and action functions for different goals, some representation of the goal must be included as input to the networks. This could be done very simply by duplicating the terms of the current state representation and using them

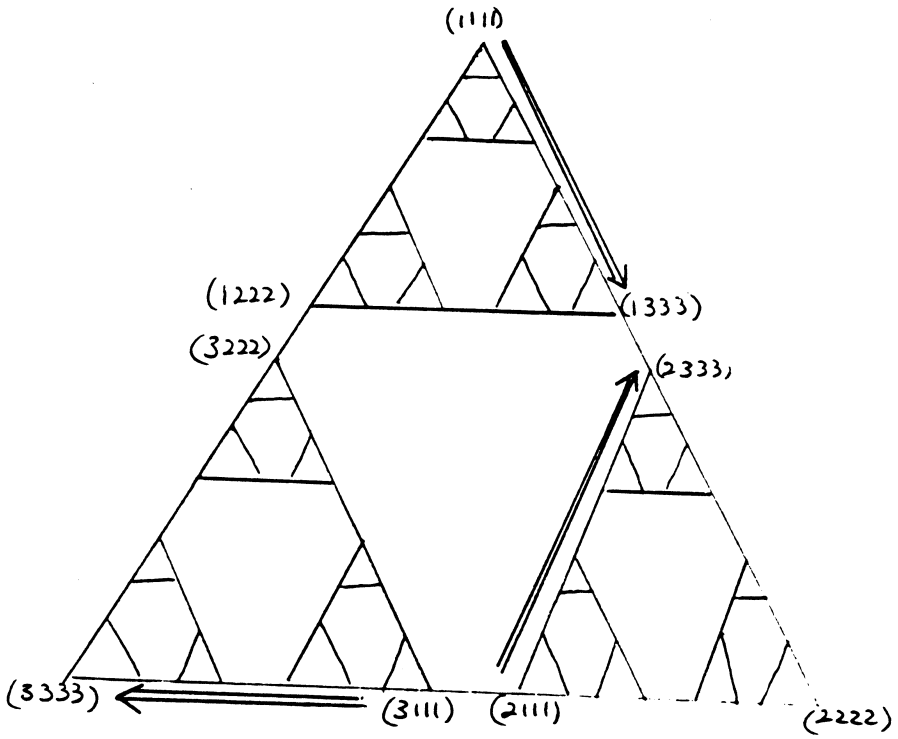


Figure 7.11: State Transition Graph for Four-Disk Tower of Hanoi Puzzle

to encode the goal state. Different evaluation and action functions would then be learned for different goals, though a multilayer action network would probably be required.

7.4 Conclusion

In Chapter VI, a connectionist learning algorithm was shown to solve a task having a *large search space*, delayed reinforcement, and requiring non-trivial (nonlinear) combinations of features. In this chapter, its generality was tested by applying it to a task with a *small search space*, requiring non-trivial feature combinations, and for which reinforcement is delayed *and* infrequent. We have shown how some of the credit-assignment techniques that have been developed for learning rules *while doing* can be incorporated into a reinforcement scheme.

The connectionist learning system was able to learn the solution to the three-disk Tower of Hanoi puzzle. The time (amount of experience) required to solve it is much greater than that required by Langley's (1985) adaptive production system, but fewer assumptions are made by the connectionist implementation. A very limited input representation was used, consisting only of the current state and the two-previous actions, though this is sufficient for a solution to be learned.

The speed of learning could be accelerated in a number of ways. The learning algorithms are just a first attempt at combining reinforcement-learning techniques with a particular method for learning in multilayer networks. The combination of reinforcement-learning with other multilayer learning algorithms should be investigated. Some multilayer learning algorithms are more appropriate when credit-assignment methods can be completely trusted, and can remove errors in a single step (such as Reilly, Cooper, and Elbaum, 1982). Such approaches can lead to a proliferation of features—in the limit one for every state. This would not be a problem for the three-disk Tower of Hanoi task, but it is not a general solution for tasks with a large number of states.

Comparisons of this connectionist approach for strategy learning with symbolic approaches highlights some of the limitations of connectionist representations. For example, the connectionist system used for the Tower of Hanoi experiments is not capable of doing variable binding in the way that Langley's (1985) production system can. Langley's system was able to learn a single symbolic rule that uses action variables to prohibit actions that are the inverses of the previous actions. Langley's production system was able to learn such rules using built-in knowledge of what "inverse" means and how particular actions and states can be generalized to variables. This raises two questions of fundamental importance to the research of connectionist systems. First, how can variables be represented in connectionist representations? This is a well-known limitation of connectionist representations (e.g., Touretzky and Hinton, 1985). In our implementation, actions are not generalized to variables; distinct negative weights from each action to its inverse had to be learned. Second, given that variables are not represented, how can task knowledge, such as the "inverse" action concept, be used to generalize one experience to other actions? For the action representation used in this chapter, this would entail some mechanism that a) observes the negative change in the weight between the previous action and its inverse, and b) uses knowledge of which weights connect other actions and their inverses to duplicate this weight change for the other actions. The answers to both questions will certainly involve connectionist structures of greater complexity than the small networks used in this study. Perhaps a connectionist implementation of symbolic processing is required (e.g., Touretzky, 1986).

Chapter 8

Summary and Future Work

The difficulty of the structural credit-assignment problem for learning in the hidden units of a connectionist system has led to a diverse set of proposed solutions. Similarities and differences in these methods are emphasized by reviewing them within a consistent framework. In this thesis, detailed performance comparisons have been made by applying several different kinds of methods for learning in hidden units to a single task. Then a connectionist approach to the learning of problem-solving strategies is developed by integrating the error back-propagation algorithm for learning in hidden units with reinforcement-learning algorithms. The ability of the connectionist system to overcome limitations of its input representation is demonstrated through its application to two strategy-learning tasks. Commonalties in the learning of new features and the learning of new symbolic terms are discussed.

8.1 Comparison of Methods for Learning in Hidden Units

Eleven hidden-unit learning algorithms were compared by applying them to the task of learning a multiplexer function. The algorithms were tested in the hidden units of a two-layer network. Two kinds of performance measures were used: the number of errors accumulated throughout a training run and the total number of input vectors for which the final weight values of a run result in an incorrect output. Care was taken to try different parameter values for each algorithm and to present performance measures as averages and confidence intervals over repetitive training runs. The results in Chapter IV are as follows.

The learning algorithm with the best performance of the algorithms compared is the back-propagation algorithm (Rumelhart, Hinton, and Williams, 1986), which is a gradient-descent procedure. Since the desired output of the system was available (given by the multiplexer function), the sample gradient of a criterion function with respect to each weight could be calculated. Next in performance are some reinforcement-learning algorithms (Barto and Anandan, 1985; Sutton, 1984), which do not require the desired output to be known but use an evaluation, or reinforcement, of the system's output to obtain an approximation of a gradient through a stochastic search of the possible output values for each unit. Standard optimization techniques were also applied by directly searching (both random and deterministic search methods were tried) for the best weight vector encompassing all hidden-unit weights. Such a large search space and the ignorance of the network's structure resulted in very poor learning performance for these methods. Surprisingly, even worse performance resulted from a heuristic error back-propagation scheme proposed by Rosenblatt (1962). This is evidence for the subtlety of the structural credit-assignment problem.

Some novel reinforcement-learning algorithms were also tested. One is an attempt to provide the hidden units with more informative, unique, reinforcement values by back-propagating reinforcements based on the weight with which the hidden units are connected to the output unit. This modification exhibits faster error reduction early in a learning run, but the rate of error reduction slows and is surpassed by that of the unmodified algorithm. This indicates that the manner in which the additional structural knowledge (the interconnection weights) is used is advantageous

for a system that has not acquired significant inter-unit weights, but is disadvantageous once approximately correct weight values have been found.

A second novel algorithm was designed as an attempt to combine reinforcement-learning methods with the following heuristic: a potentially useful feature is one that discriminates among input vectors for which the system is performing poorly, or that discriminates these input vectors from others for which the system's performance is good. The addition of features with these properties provides to the output layer new degrees of freedom with which it can learn to generate more appropriate output values. To learn such features, the weight vector of a hidden unit is shifted towards or away from the current input, depending on whether the resulting reinforcement value is low or high, respectively. A gradual transition from this process to the original reinforcement-learning algorithm takes place for each hidden unit as the unit develops an output weight of increasing magnitude. This results in a faster reduction of errors, bringing the cumulative number of errors for this augmented reinforcement-learning algorithm closer to that of the error back-propagation method.

These results are summarized in Figure 4.5, which is a superposition of learning curves for the learning algorithms that were tested. The recent error back-propagation and reinforcement-learning algorithms perform much better than the direct search methods and Rosenblatt's method. A recent contribution that was not included in the comparative study is the Boltzmann Machine learning algorithm (Ackley, Hinton, and Sejnowski, 1985). Time did not allow its inclusion in the current study, so comparisons with this algorithm are reserved for future work.

As in all empirical studies, it is important to stress that the results are valid only for the particular task and training regime that was used during the experiments. For example, a task requiring a smaller network might be most readily solved by a random search of the entire weight space. In fact, in selecting a task for the comparative study, a small task with two input components was tested and it was discovered that a random search solved the task faster than the error back-propagation and reinforcement-learning algorithms. The multiplexer task was chosen because the weight space was sufficiently large that direct optimization methods are slow, and also because it wasn't so complex that a prohibitive amount of simulation time would be needed to gather significant performance statistics. Time also prohibited the extension of the comparative study to other, more complex tasks as would be required to address issues regarding how well the algorithms scale up to harder tasks.

8.2 Strategy Learning with Multilayer Connectionist Systems

Strategy learning can be characterized as the acquisition of a method for generating actions that cause desired transitions among the states of a problem. The desirability of particular transitions is often indicated by an evaluation that imposes a preference ordering on the possible transitions from a given state. Reinforcement-learning methods have been shown to be a viable approach to learning to select the best action under these conditions (Barto, Sutton, and Brouwer, 1981; Barto and Sutton, 1981a), whereas most connectionist learning methods are based on knowledge of the correct action.

For some tasks, an evaluation is not immediately available, but occurs only after a sequence of actions has been generated. Sutton (1984) has developed the AHC algorithm for dealing with this temporal credit-assignment problem. Hampson (1983) has develop a similar, though less general, algorithm. Barto, Sutton, and Anderson (1983; Sutton, 1984) combined the AHC algorithm with a reinforcement-learning algorithm into a single-layer connectionist system for strategy learning. A major accomplishment of this thesis is the extension of their learning algorithms for single-layer systems to algorithms for learning strategies with multilayer connectionist systems.

Reinforcement-learning has been successfully applied to learning in the hidden units of multilayer connectionist systems (Barto, Anderson, and Sutton, 1982; Barto and Anderson, 1985; Anderson, 1982; Barto, 1985), but since the error back-propagation algorithm resulted in faster error reduction for the multiplexer task, this algorithm was combined with the AHC and the reinforcement-learning algorithms. This hybrid system was applied to two tasks that were difficult, because

- the functions to be learned are nonlinear, so a single-layer system given the original features is not sufficient, and
- a performance evaluation appears only after a sequence of actions have been taken, making it difficult to identify which actions are good and which are bad.

One task involved the control of a dynamical system—a simulated pole that was to be balanced for as long as possible. The second task was chosen to demonstrate the application of the fundamentally numerical, connectionist approach to a task that has typically been solved with symbolic representations. The Towers of Hanoi puzzle was used to facilitate comparisons with published results from the application of an adaptive production system to this puzzle (Langley, 1985).

8.2.1 Pole Balancing

The major difficulty of the pole-balancing task is due to the use of a very uninformative evaluation signal. Task-specific information, such as the dynamics of the pole, were assumed to be unavailable to the design of the learning system. The evaluation signal was supplied only when the pole fell or hit the end of the track, so other information concerning the task objective, like the advantage of maintaining the pole near the vertical, was not assumed. Of course, if such information is available it should be incorporated into the initial design of a learning system. Our intentions were to develop learning algorithms for those parts of a task for which a minimum amount of information is available.

The combination of the error back-propagation algorithm with the AHC and reinforcement-learning algorithms was successful: the two-layer system balanced the pole for many more steps than did a one-layer system receiving the same representation of the pole's state. The hidden units learned features with which the output units could overcome the limitations imposed by the representation of the pole's state and the linearity of the output functions. In analyzing the new features that were formed, it was discovered that only a small number of new features were needed to solve the task. Some runs resulted in the formation of a single new feature, while others resulted in up to three features that developed significant influence on the system's output. For two of the ten runs using the two-layer system, the pole was balanced for approximately 9 simulated minutes, at which point the runs were terminated.

Our previous experiments (Barto, Sutton, and Anderson, 1983) with the pole-balancing task involved a single-layer system for which each state of the pole was represented by a 162-dimensional standard unit basis vector. The continuous state space was discretized into 162 distinct, 4-dimensional rectangles to allow the system's units, whose outputs are based on linear weighted sums of input, to learn the desired functions. The current connectionist system differs in the absence of this "decoder" and the addition of hidden units that through experience learn features that decode the state into an appropriate form. Another difference, which makes performance comparisons difficult, is that after every failure the state of the pole is set to a random state instead of the zero state (vertical, stationary pole), as was done in the previous experiments. For this reason, many more failures were generated in the current paradigm, because some reset states were very near failure states. We can say that after the same number of training steps, the current system had not attained as high an average balancing time as had the previous system. This is due to either a) the additional experience needed to learn useful new features, or b) the lack of experience in critical states (such as the zero state) for which nearby states require opposite actions.

8.2.2 Tower of Hanoi

The generality of the connectionist system was tested by applying it with few modifications to a non-numerical problem-solving task—the Towers of Hanoi puzzle. Similar restrictions on the amount of a priori knowledge were assumed; primarily a final reinforcement at the end of a successful sequence of actions (as opposed to an unsuccessful sequence for the pole-balancing task) provided information regarding the objective of the task. The state of the puzzle is presented to the connectionist system as binary-valued features representing the peg on which each disk resides.

The two-layer system again performed better than a single-layer system. The two-layer system reliably found a minimum-length solution, i.e., the system applied a sequence of actions consisting of the minimum number of actions required to achieve the goal state. In solving the puzzle, an evaluation function was learned that ranked states according to the smallest number of moves between the state and the goal state.

In learning an evaluation function, a number of new features were developed. In the state-transition graph for the Towers of Hanoi puzzle there are several bottlenecks—parts of the graph are interconnected by a single path. New features were formed that discriminate states in the bottlenecks from other states. The output unit of the evaluation network could not learn a monotonically-increasing function through the bottlenecks with the original features, but the new features resulted in a good evaluation function.

Langley (1985) developed an adaptive production system that learned to solve the Towers of Hanoi puzzle, plus several others. The connectionist system that we applied to the Towers of Hanoi puzzle has few similarities to Langley's production system. It is instructive to analyze the differences and to question whether or not they are fundamental distinctions between symbolic and connectionist representations.

One difference is that Langley uses a full history of past states and actions to aid the assignment of credit, whereas the connectionist system relies on the learning of a good evaluation function to solve the credit assignment problem. This difference is not fundamental to the representations involved; evaluation functions can be used for symbolic systems and a history of states and actions can be of use in training a connectionist system. A history could be used much as it is for the symbolic system, by retrieving the events as training instances. A separate issue is the association by a connectionist system of past events with current action probabilities in order to base decisions on previous states and actions. In our experiments, the connectionist system does receive the two previous actions as input, so two and three step sequences can be learned; the inclusion of all past events as input to the system is not feasible. An alternative is to collapse the history into a weighted average of past events (Jordan, 1986).

Another difference is that a breath-first search is not performed by the connectionist system. In its initial, naive state, the connectionist system chooses actions randomly and as the evaluation function develops, the action probabilities become increasingly biased towards actions that result in state transitions producing positive changes in the evaluation function. Breadth-first search control can be added to the connectionist framework by disregarding the probabilistic generation of actions and presenting state-action pairs as training instances after some process has assigned credit to every pair. Learning an evaluation function in this case requires the extraction of desirable paths from a state history. One attraction of the connectionist approach demonstrated here is its ability to learn with minimal resources for search control and history maintenance.

A very important distinction that is currently a topic of debate is the use of *variables*. A single symbolic rule can be applied to many situations through the binding of variables. For example, Langley's system learns a rule that, through variable binding, can be used to avoid the application of the previous action's inverse for all possible actions. With one training instance and knowledge of what an action's "inverse" means, a single rule is learned that generalizes to all other actions. It is not clear how knowledge of an action's inverse can be used in a connectionist system to either a) learn the connectionist analog of a generalized rule with variables, or b) duplicate the weight changes due to experience with one action to the weights of other actions. Touretzky and Hinton (1985) have shown how variable binding can be performed in a particular connectionist system.

Related to the issue of variables is the issue of the transfer of learning. After learning strategies for solving one task, an efficient learning system must be able to exploit common aspects between this task and subsequent tasks by applying in similar situations the strategies that worked well for the first task. Langley's production rule having a variable action and state can be immediately applied to other, more complex Tower of Hanoi puzzles. This is not possible for the connectionist system and the state and action representations used here. Learning is transferred but not to the degree possible with variablized rules. The strategies learned from the 3-disk Tower of Hanoi are specifically dependent on the 3 disks—the addition of another disk does not affect the strategies until further learning occurs. Different representation of states and actions would result in different amounts of transfer.

In addition to transfer to larger tasks, we can consider transfer to tasks having different goal states. A representation of a task's goal can be included in a connectionist network's input

representation and therefore acquire an effect, through learning, on the strategies followed in solving a problem. Again, the amount of transfer depends on the representation of the goal.

8.3 Future Work

Learning algorithms now exist for multilayer connectionist systems, but their convergence to solutions is slow. Methods for accelerating the convergence of the algorithms warrant further study (Saridis, 1970; Barto and Sutton, 1981b; Parker, 1985) In addition, ways of breaking a gradient-following search out of local minima, such as the combination of the A_{R-P} algorithm with a data-directed search described in Chapter IV, could reduce learning time and increase the chances of finding solutions.

The development of future learning algorithms must be related to the issues of generalization, i.e., how learning in one situation is transferred to similar situations. To take full advantage of the generalization properties of connectionist systems, the search for new features must be guided by consideration of how they will affect the transfer of learning in future situations, as discussed in Chapter II. Such methods could be developed into theories of how animals "learn to learn" and that account for behaviors from "learning set" experiments (Kehoe, 1986). Transfer of learned responses does not account for these behaviors; rather, the acquisition of a representation tailored to the learning of particular tasks seems to occur.

Ways of extending the capabilities of the simple connectionist systems that have been dealt with to date are required. The ability to scale-up current architectures and algorithms to large, realistic problems requires further research on the learning of multilayer, hierarchical representations. Concerns similar to those illustrated by Fu and Buchanan (1985) of learning intermediate levels of knowledge must be addressed in connectionist learning research.

8.4 Applications

Connectionist schemes have potential application for domains requiring complex decision-making. The problem of programming a connectionist network to perform as desired can be intractable; complex decisions would require the specification of a great number of interconnection weights. With the recent advent of successful multilayer learning algorithms, connectionist systems appear more feasible.

Chapter VI demonstrated an application to a numerical control task. This task is representative of a wide class of tasks involving the maintenance of balance, such as the control of rocket thrusters and of walking machines. An even larger class of control tasks, including those just mentioned, are tasks with delayed evaluation signals. For example, in a process control situation, the effects of adjustments to process parameters may not be observed immediately. A system like that used in Chapter VI to predict failure could be used to appropriately credit a parameter change. The "generate-and-test" paradigm, however, could only be used when the cost of an incorrect action is small.

The solution of the Towers of Hanoi problem in Chapter VII shows that connectionist methods can be applied to tasks with a symbolic flavor. Problem-solving and planning tasks involving a large number of states might be more practically addressed by the connectionist approach of not explicitly storing a history of past experience with a task. A more reasonable application of connectionist systems would be in combination with symbolic techniques, taking advantage of symbolic reasoning methods and the generalization and fast decision-making abilities of connectionist representations.

We are just beginning to see the potential of connectionist representations in machine learning applications. A common criticism of connectionist systems—that the difficulty of learning in multilayer networks restricts their usefulness—has been diminished by the recent development of techniques for learning in hidden units. Many difficult problems remain, but we hope that the

work presented here contributes to an understanding of the issues of connectionist learning and to the extension of the connectionist paradigm to the learning of problem-solving strategies.

Appendix A

Results from One-Layer Network Experiments of Chapter II

This appendix contains results from parameter sensitivity investigations for the single-layer connectionist systems of Chapter II. These results are discussed in Chapter II, so they are simply listed here with little explanation.

Due to the large number of parameter values that were tested, the results of the experiments with the output-vector task are presented in graph form. The 99% confidence intervals are shown as vertical bars superimposed on every data point.

Original Representation

ρ	ν	μ
0.0001	23.1 ±1.2	20,371 ±35
0.001	24.4 ±1.7	20,406 ±37
0.01	24.0 ±1.3	20,383 ±32
0.1	23.3 ±1.7	20,379 ±43
1	24.6 ±1.9	20,379 ±39
10	23.9 ±1.3	20,357 ±43

Basis Vector Representation

ρ	ν	μ
0.001	0.0 ±0.0	32.0 ±0.0
0.01	0.0 ±0.0	32.0 ±0.0
0.1	0.0 ±0.0	32.0 ±0.0
1	0.0 ±0.0	32.0 ±0.0

Original Representation Plus New Features

ρ	ν	μ
0.001	0.0 ±0.0	31.5 ±2.3
0.01	0.0 ±0.0	30.6 ±2.0
0.1	0.0 ±0.0	32.8 ±2.1
1	0.0 ±0.0	31.8 ±2.5
10	0.0 ±0.0	30.8 ±2.5

Table A.1: Multiplexer Task—Parameter Search Results

Original Representation

ρ	μ_1	μ_2
0.0001	21.9 ±0.14	21.9 ±0.11
0.001	22.0 ±0.05	21.9 ±0.14
0.01	22.0 ±0.05	21.9 ±0.16
0.1	22.0 ±0.07	22.0 ±0.07
1	22.0 ±0.05	22.0 ±0.00
10	21.9 ±0.11	21.9 ±0.11

Basis Vector New Features

ρ	μ_1	μ_2
0.0001	21.9 ±0.14	4.0 ±0.00
0.001	22.0 ±0.05	4.0 ±0.00
0.01	22.0 ±0.05	4.0 ±0.00
0.1	22.0 ±0.07	4.0 ±0.00
1	22.0 ±0.05	4.0 ±0.00
10	21.9 ±0.11	4.0 ±0.00

New Common Features

ρ	μ_1	μ_2
0.0001	21.9 ±0.14	2.9 ±0.28
0.001	22.0 ±0.05	2.9 ±0.29
0.01	22.0 ±0.05	3.1 ±0.27
0.1	22.0 ±0.07	3.1 ±0.28
1	22.0 ±0.05	3.0 ±0.25
10	21.9 ±0.11	3.3 ±0.29

Table A.2: Input-Cluster Task—Parameter Search Results

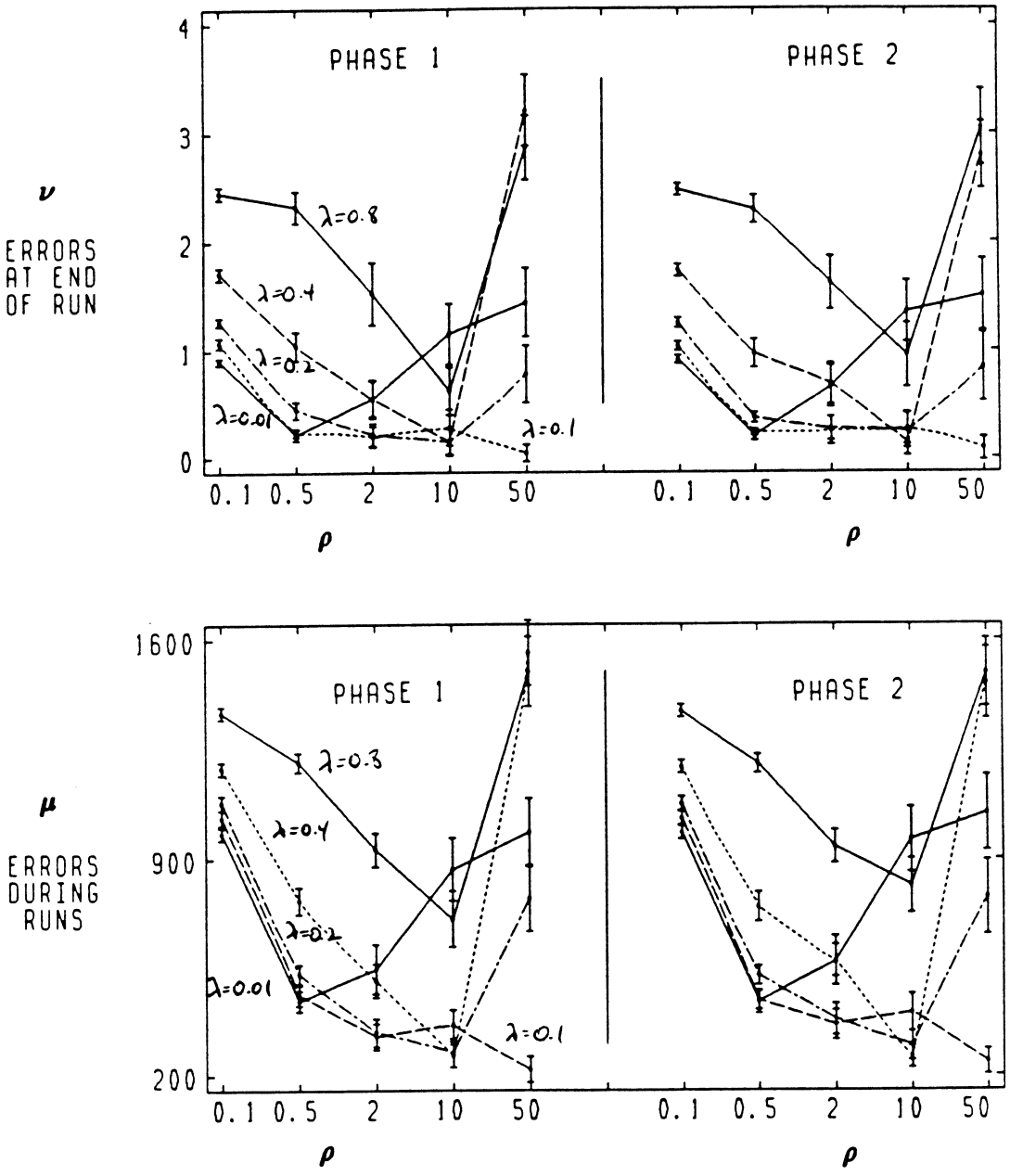


Figure A.1: Output-Vector—Original Representation

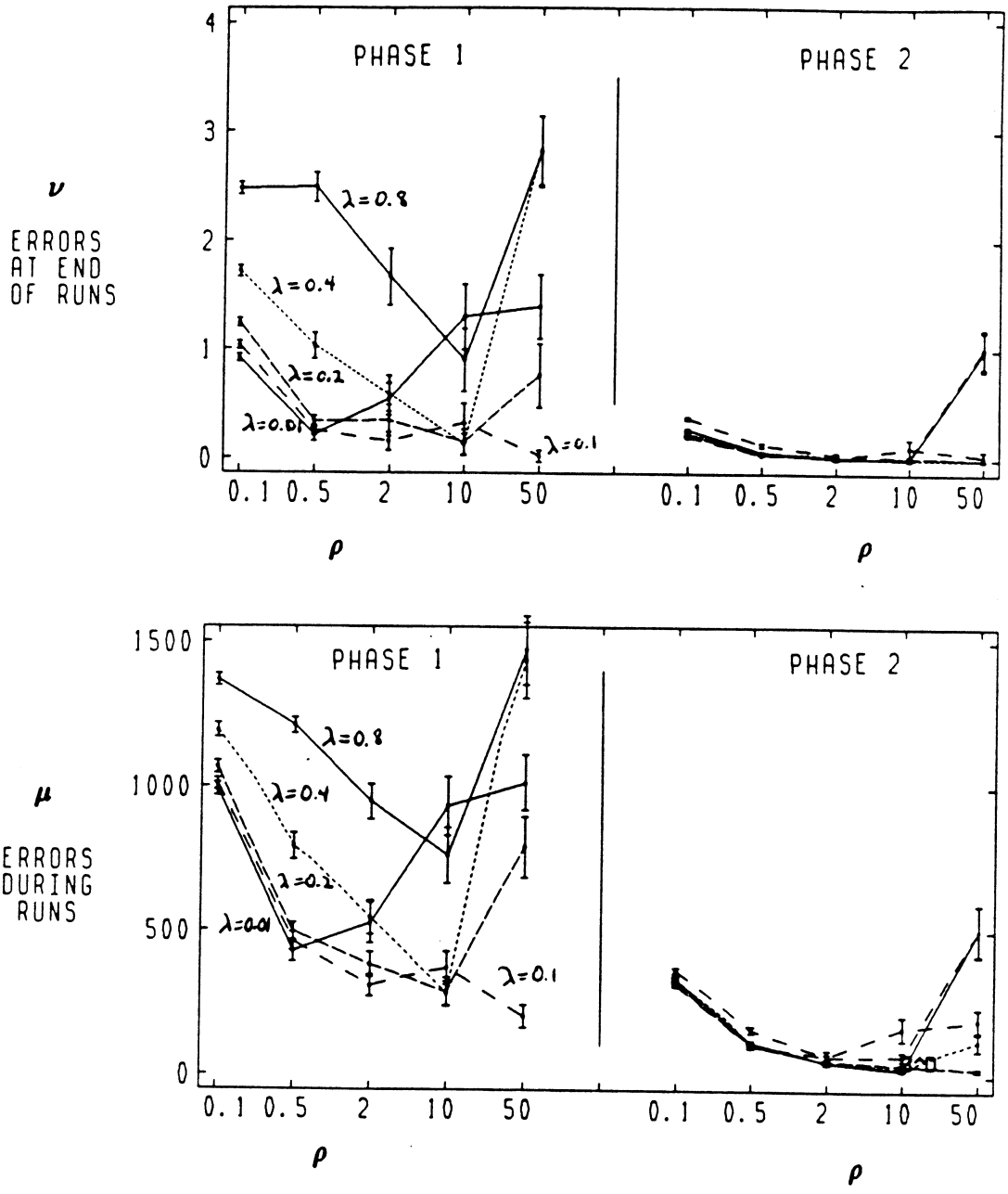


Figure A.2: Output-Vector—New Output Representation

Appendix B

Derivation of $P(a_j = 1)$ for Tower of Hanoi Experiments

The output values of the action network for the Tower of Hanoi experiments of Chapter VI are determined by setting the output of the unit with the largest weighted sum plus noise to 1 and the other outputs to 0. Only those units representing legal actions for the current state are involved in this competition. For a given state there are either two or three legal actions, so the competition on any step is between two or three units. This simplifies the calculation of $P(a_j = 1)$, since at most the output of three units must be considered in determining the probability of the output of one unit being the maximum. To derive an expression for $P(a_j = 1)$, we must solve problems concerning relationships between the values of two or three random variables, the random variables being the weighted sums of the units plus noise.

B.1 Two Legal Actions

We start with the case for which only two actions, a_{j_1} and a_{j_2}, can be legally applied to the current state. First we define variable $s_j[t]$ as the weighted sum of Unit j's input:

$$s_j[t] = \sum_{i=0}^{21} w_{ij}[t]\, x_i[t].$$

Now the random variables X and Y are defined for Units j_1 and j_2, respectively, to be the units' weighted sums plus noise:

$$X = s_{j_1}[t] + \eta_{j_1},$$

$$Y = s_{j_2}[t] + \eta_{j_2},$$

where the η_j are random variables from an exponential distribution. Let X_m be the minimum value of X and Y_m be the minimum value of Y, which are simply the weighted sums:

$$X_m = s_{j_1}[t],$$

$$Y_m = s_{j_2}[t].$$

The density functions for random variables X and Y are given by:

$$f_X(x) = e^{X_m - x}$$

Case	Integral Expression	Evaluation
$X_m < Y_m$	$\int_{Y_m}^{\infty} f_Y(y) \int_{y}^{\infty} f_X(x)\, dx\, dy$	$\frac{e^{X_m - Y_m}}{2}$
$X_m \geq Y_m$	$\int_{X_m}^{\infty} f_X(x) \int_{Y_m}^{x} f_Y(y)\, dy\, dx$	$1 - \frac{e^{Y_m - X_m}}{2}$

Table B.1: Expressions for $P(X > Y)$ by Cases

$$f_Y(y) \quad = \quad e^{Y_m - y}$$

An equation for $P(a_j = 1)$ is derived by isolating the cases of X_m or Y_m being the larger, and finding expressions for $P(X > Y)$ for each case. Integrals over the possible values of X and Y result in the expressions shown in Table B.1. A single equation for $P(a_{j_1}[t] = 1)$, and therefore for $E\{a_{j_1}[t] \,|\, w; x\}$, when there are two legal actions for the current state is given by:

$$E\{a_{j_1}[t] \,|\, w; x\} = \begin{cases} \frac{1}{2} e^{s_{j_1}[t] - s_{j_2}[t]}, & \text{if } s_{j_1}[t] < s_{j_2}[t]; \\ 1 - \frac{1}{2} e^{s_{j_2}[t] - s_{j_1}[t]}, & \text{if } s_{j_1}[t] \geq s_{j_2}[t]. \end{cases}$$

B.2 Three Legal Actions

For three legal actions, a_{j_1}, a_{j_2}, and a_{j_3}, we must define an additional random variable:

$$Z = s_{j_3}[t] + \eta_{j_3},$$

with lower bound:

$$Z_m = s_{j_3}[t].$$

Z also has the exponential density function

$$f_Z(z) = e^{Z_m - z}.$$

There are six possible orderings of X_m, Y_m, and Z_m, and each is analyzed in turn to determine $P(X > Y, X > Z)$. The resulting expressions for $P(X > Y, X > Z)$ for each case are shown in Table B.2. These are combined into the following expression for the expected value of $a_{j_1}[t]$ (t is dropped for clarity):

$$E\{a_{j_1} \,|\, w; x\} = \begin{cases} 1 - \frac{1}{2} e^{s_{j_2} - s_{j_1}} - \frac{1}{2} e^{s_{j_3} - s_{j_1}} \\ \qquad + \frac{1}{3} e^{s_{j_2} + s_{j_3} - 2 s_{j_1}}, & \text{if } s_{j_1} \geq s_{j_2},\ s_{j_1} \geq s_{j_3}; \\ \frac{1}{2} e^{s_{j_1} - s_{j_2}} - \frac{1}{6} e^{s_{j_1} + s_{j_3} - 2 s_{j_2}}, & \text{if } s_{j_2} \geq s_{j_1},\ s_{j_2} \geq s_{j_3}; \\ \frac{1}{2} e^{s_{j_1} - s_{j_3}} - \frac{1}{6} e^{s_{j_1} + s_{j_2} - 2 s_{j_3}}, & \text{if } s_{j_3} \geq s_{j_1},\ s_{j_3} \geq s_{j_2}; \end{cases}$$

Case	Integral Expression	Evaluation
$X_m \geq Y_m \geq Z_m$	$\int_{X_m}^{\infty} f_X(x) \int_{Y_m}^{x} f_Y(y) \int_{Z_m}^{x} f_Z(z)\, dz\, dy\, dx$	$1 - \frac{1}{2}e^{Y_m - X_m}$ $- \frac{1}{2}e^{Z_m - X_m}$ $+ \frac{1}{3}e^{Y_m + Z_m - 2X_m}$
$X_m \geq Z_m \geq Y_m$	$\int_{X_m}^{\infty} f_X(x) \int_{Z_m}^{x} f_Z(z) \int_{Y_m}^{x} f_Y(y)\, dy\, dz\, dx$	$1 - \frac{1}{2}e^{Z_m - X_m}$ $- \frac{1}{2}e^{Y_m - X_m}$ $+ \frac{1}{3}e^{Z_m + Y_m - 2X_m}$
$Y_m \geq X_m \geq Z_m$	$\int_{Y_m}^{\infty} f_X(x) \int_{Y_m}^{x} f_Y(y) \int_{Z_m}^{x} f_Z(z)\, dz\, dy\, dx$	$\frac{1}{2}e^{X_m - Y_m}$ $- \frac{1}{6}e^{X_m + Z_m - 2Y_m}$
$Z_m \geq X_m \geq Y_m$	$\int_{Z_m}^{\infty} f_X(x) \int_{Y_m}^{x} f_Y(y) \int_{Z_m}^{x} f_Z(z)\, dz\, dy\, dx$	$\frac{1}{2}e^{X_m - Z_m}$ $- \frac{1}{6}e^{X_m + Y_m - 2Z_m}$
$Y_m \geq Z_m \geq X_m$	$\int_{Y_m}^{\infty} f_X(x) \int_{Y_m}^{x} f_Y(y) \int_{Z_m}^{x} f_Z(z)\, dz\, dy\, dx$	$\frac{1}{2}e^{X_m - Y_m}$ $- \frac{1}{6}e^{X_m + Z_m - 2Y_m}$
$Z_m \geq Y_m \geq X_m$	$\int_{Z_m}^{\infty} f_X(x) \int_{Y_m}^{x} f_Y(y) \int_{Z_m}^{x} f_Z(z)\, dz\, dy\, dx$	$\frac{1}{2}e^{X_m - Z_m}$ $- \frac{1}{6}e^{X_m + Y_m - 2Z_m}$

Table B.2: Expressions for $P(X > Y, X > Z)$ by Cases

Appendix C

Weights for Solutions of Strategy Learning Tasks

The values of the weights at the end of the particular runs discussed in the text of Chapters VI and VII are included in this appendix for those who wish to study the behavior of the resulting systems in more detail. The values are relatively arranged as they appear in the network schematics pictured in the figures of the above chapters. Weights for the single-layer networks and for the two-layer networks are shown. First the resulting weights of the pole-balancing runs are presented, followed by the weights for the Tower of Hanoi runs.

C.1 Pole-Balancing Task

Figure C.1a shows the weights for the one-layer evaluation network and Figure C.1b shows the one-layer action network's weights. These figures correspond to the network schematic in Figure 6.4. Figure C.2a and C.2b are the weights for the two-layer networks, corresponding to the network schematics of Figure 6.7.

C.2 Tower of Hanoi Task

Now the weights from runs with the Tower of Hanoi task are presented. Figure C.3a and C.3b are the weights for the single-layer evaluation and action networks, respectively. This figure corresponds to Figure 7.5. Weights for the two-layer evaluation network with the one-layer action network are divided into two figures. Figure C.4 shows the weights for the two-layer evaluation network and Figure C.5 shows the weights for the one-layer action network. The corresponding figure in Chapter VII is Figure 7.8.

−0.22	−6.76
−0.00	−16.00
−0.09	−1.75
0.02	15.61
−0.02	5.92

Figure C.1: Pole-Balancing—Weights Learned by One-Layer Network

5.69	5.60	−0.16	5.64	5.62	−3.06	3.11	3.16	0.11	3.10	3.10	−5.24
3.45	3.53	1.64	3.54	3.43	−0.34	2.19	2.75	2.91	2.82	2.82	15.82
3.32	3.34	1.22	3.25	3.40	−0.53	2.41	1.39	−2.84	1.63	1.46	3.70
1.96	1.92	2.30	1.79	1.87	−0.71	1.23	1.50	5.03	1.45	1.54	28.38
3.28	3.33	1.42	3.28	3.31	−0.58	2.34	2.71	4.11	2.69	2.73	14.13
					−1.00						−8.98
					−1.01						−9.03
					6.95						9.44
					−1.00						−8.94
					−1.03						−8.59

Figure C.2: Pole-Balancing—Weights Learned by Two-Layer Network

−0.02	−1.83	2.47	−2.56	7.36	−0.72	−4.73
−0.35	3.52	2.83	−6.18	−3.29	−2.49	5.61
−0.22	−12.01	−8.85	−0.76	18.20	4.67	−1.25
0.54	4.84	10.97	1.82	−0.18	−3.62	−13.83
−0.12	−13.76	−10.33	−0.67	19.45	−0.29	5.59
−0.01	1.32	8.12	−6.17	−2.69	−0.73	0.15
0.09	8.78	7.16	1.73	−2.04	−0.42	−15.21
0.08	−5.69	−7.62	0.00	21.91	0.00	−8.61
−0.03	0.00	12.57	−4.24	−7.18	−1.14	0.00
−0.08	2.03	0.00	−0.87	0.00	−0.29	−0.86
	0.39	−1.01	0.80	0.13	−0.44	0.14
	1.04	7.90	−0.83	−2.57	2.98	−8.52
	−10.66	−8.41	0.82	21.16	−1.25	−1.66
	3.64	1.95	0.16	−0.23	−3.97	−1.55
	−2.68	0.25	−1.90	−1.30	2.55	3.08
	4.03	4.85	−4.16	−2.46	−1.30	−0.96
	0.93	8.30	−5.01	−3.85	−0.37	0.00
	−7.43	−12.40	0.57	20.71	0.00	−1.44
	2.99	8.41	−0.02	0.00	0.65	−12.03
	−6.84	−2.54	0.00	0.78	4.14	4.47
	6.70	−0.30	2.47	−0.59	−8.02	−0.25
	0.00	3.48	−3.12	−2.32	2.16	−0.20

Figure C.3: Tower of Hanoi—Weights Learned by One-Layer Networks

−2.63	−1.31	−2.97	−0.20	−1.70	0.00	−2.05	3.24	1.89	0.02	0.33
−5.00	−3.17	−0.14	1.43	−2.54	0.85	−2.11	3.06	−2.92	1.47	−0.14
2.47	1.13	−3.19	−2.81	2.38	2.10	−1.02	−0.38	1.97	−4.45	−0.18
−2.29	−1.80	−2.53	1.66	−4.50	−1.93	−0.81	3.62	4.55	3.56	0.98
2.56	−4.76	−2.46	−3.18	−3.13	3.92	−0.26	0.47	−0.83	3.45	0.23
−3.09	1.42	−3.37	2.15	0.16	−6.12	−2.51	2.21	0.25	3.60	−0.01
−3.84	−0.01	1.19	0.37	−0.70	2.98	0.01	4.11	4.49	−6.67	0.43
−2.77	−4.11	−6.38	−6.52	−3.97	2.01	−2.27	−0.06	2.86	−1.30	0.62
0.59	4.88	1.66	2.58	−0.45	−1.04	−0.32	3.22	0.17	3.54	−0.16
−2.03	−4.11	−0.94	4.21	−0.31	−0.80	−1.39	3.09	1.11	−1.55	0.19

−0.26
−1.08
0.09
0.14
−0.48
0.32
0.20
0.20
0.43
1.03

Figure C.4: Tower of Hanoi—Weights Learned by Two-Layer Evaluation Network

−0.33	0.52	0.05	0.31	−0.47	−0.08
0.89	1.70	−1.04	−1.28	−1.80	1.52
−0.55	−1.17	0.84	−0.17	1.14	−0.10
−1.00	0.51	0.30	2.06	−0.28	−1.59
0.33	−0.41	0.38	0.75	−0.52	−0.53
−0.60	0.76	0.24	−0.77	−0.30	0.68
−0.38	0.71	−0.52	0.63	−0.12	−0.32
−2.40	0.05	0.00	1.98	0.00	0.37
0.00	1.00	0.88	−1.37	−0.52	0.00
1.74	0.00	−0.78	0.00	−0.43	−0.54
−0.35	−0.46	1.10	−0.34	−0.43	0.46
0.01	−0.01	−0.33	0.10	0.15	0.08
−0.49	−0.55	−0.35	1.37	−0.53	0.56
0.91	0.46	−0.22	−0.83	0.03	−0.33
1.10	−0.42	−0.65	0.52	−0.15	−0.40
0.43	−0.22	0.56	−0.20	−0.01	−0.55
0.49	2.51	−0.52	−2.65	0.17	0.00
0.72	−3.05	0.59	1.84	0.00	−0.10
0.22	0.34	0.14	0.00	1.09	−1.79
−2.92	−1.03	0.00	1.69	1.32	0.94
0.83	2.13	1.17	−0.60	−3.12	−0.41
0.00	0.14	−1.28	0.34	−0.40	1.20

Figure C.5: Tower of Hanoi—Weight Values Learned by One-Layer Action Network with Two-Layer Evaluation Network

Bibliography

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

Albus, J. S. (1981) *Brains, behavior, and robotics.* BYTE Publications.

Alder, M. D. (1975) A convergence theorem for hierarchies of model neurones. *SIAM Journal of Computing*, *4*, 192–201.

Amarel, S. (1981) Problems of Representation in Heuristic Problem Solving: Related Issues in the Development of Expert Systems. *Technical Report CBM-TR-118*, Laboratory for Computer Science Research, Rutgers University, New Brunswick, NJ.

Anderson, C. W. (1982) Feature generation and selection by a layered network of reinforcement learning elements: some initial experiments. *Technical Report COINS 82–12*, University of Massachusetts, Amherst.

Anzai, Y. and Simon, H. A. (1979) The Theory of Learning by Doing. *Psychological Review*, *86*.

Ballard, D. H. (1984) Parameter nets. *Artificial Intelligence*, *22*, 235–267.

Barr, A. and Feigenbaum, E. A. (1981) *The handbook of artificial intelligence, Volume 1.* Los Altos, California: Kauffman.

Barto, A. G. (1985) Learning by statistical cooperation of self-interested neuron-like computing elements. *Human Neurobiology*, *4*, 229–256.

Barto, A. G. and Anandan, P. (1985) Pattern recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, *15*, 360–375.

Barto, A. G. and Anderson, C. W. (1985) Structural learning in connectionist systems. *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, Irvine, CA.

Barto, A. G., Anderson, C. W., and Sutton, R. S. (1982) Synthesis of nonlinear control surfaces by a layered associative search network. *Biological Cybernetics*, *43*, 175–185.

Barto, A. G. and Sutton, R. S. (1981a) Landmark learning: an illustration of associative search. *Biological Cybernetics*, *42*, 1–8.

Barto, A. G. and Sutton, R. S. (1981b) Goal seeking components for adaptive intelligence: An initial assessment. *Air Force Wright Aeronautical Laboratories/Avionics Laboratory Technical Report AFWAL-TR-81-1070*, Wright-Patterson AFB, Ohio.

Barto, A. G., Sutton, R. S., and Anderson, C. W. (1983) Neuronlike elements that can solve difficult learning control problems. *IEEE Trans. on Systems, Man, and Cybernetics*, *13*, 835–846.

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

Bekey, G. A. and Masri, S. F. (1983) Random search techniques for optimization of nonlinear systems with many parameters. *Mathematics and Computers in Simulation*, *25*, 210–213.

Bush, R. R. and Estes, W. K., eds. (1959) *Studies in Mathematical Learning Theory*, Stanford University Press.

Cannon, R. H., Jr. (1967) *Dynamics of Physical Systems.* McGraw-Hill, Inc.

Cohen, P. and Feigenbaum, E. A. (1981) *The handbook of artificial intelligence, Volume 3.* Los Altos, California: Kauffman.

Duda, R. O. and Hart, P. E. (1973) *Pattern Classification and Scene Analysis.* New York: Wiley.

Duffy, J. Q. and Franklin, M. A. (1975) A learning identification algorithm and its application to an environmental system. *IEEE Transactions on Systems, Man, and Cybernetics*, *15*.

Farley, B. G., and Clark, W. A. (1954) Simulation of self-organizing systems by digital computer. *I.R.E. Transactions on Information Theory*, *4*, 76–84.

Fikes, R. E., Hart, P., and Nilsson, N. J. (1972) Learning and executing generalized robot plans. *Artificial Intelligence*, *3*, 251–288.

Fu, King-Sun, (1970) Learning control systems—review and outlook. *IEEE Transactions on Automatic Control*.

Fu, Li-Min and Buchanan, B. G. (1985) Learning intermediate concepts in constructing a hierarchical knowledge base. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*.

Fukushima, K. (1973) A model of associative memory in the brain. *Kybernetic*, *12*, 58–63.

Fukushima, K. (1980) Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, *36*, 193–202.

Gallant, S. (1985) Automatic generation of expert systems from examples. *Proceedings of Second International Conference on Artificial Intelligence Applications, IEEE Computer Society*, Miami Beach, Florida.

Gill, P. E., Murray, W., and Wright, M. H. (1981) *Practical Optimization*, Academic Press, New York.

Gillies, A. M. (1985) Machine learning procedures for generating image domain feature detectors. University of Michigan Ph. D. Dissertation.

Gilstrap, L. O., Jr. (1971) Keys to developing machines with high-level artificial intelligence. *Design Engineering Conference, ASME*, New York.

Goldberg, D. (1983) Computer aided gas pipeline operation using genetic algorithms and rule learning. University of Michigan Ph. D. Dissertation.

Hampson, S. (1983) A neural model of adaptive behavior. University of California, Irvine, Ph. D. Dissertation.

Higdon, D. T. and Cannon, R. H., Jr. (1963) On the control of unstable multiple-output mechanical systems. In *ASME Winter Annual Meeting*, Philadelphia, Pa.

Hinton, G. E., and Anderson, J. (1981) *Parallel models of associative memory.* Hillsdale, N. J.: Erlbaum.

Hinton, G. E., and Sejnowski, T. J. (1983) Optimal perceptual inference. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Washington, D.C.

Holland, J. H. (1975) *Adaptation in Natural and Artificial Systems*, University of Michigan Press.

Holland, J. H. (1986) Escaping brittleness: the possibilities of general purpose learning algorithms applied to parallel rule-based systems. In *Machine Learning: An Artificial Intelligence Approach, Volume II*, ed. by Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., Morgan Kaufmann, Los Altos, CA.

Ivanhenko, A. G. (1971) Polynomial theory of complex systems. *IEEE Transactions on Systems, Man, and Cybernetics*, *1*.

Jarvis, R. A. (1975) Optimization strategies in adaptive control: a selective survey. *IEEE Transactions on Systems, Man, and Cybernetics*, *5*, 83–94.

Kehoe, E. J. (1986) A layered network model for learning-to-learn and configuration in classical conditioning. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, Amherst, MA., pp. 154–175.

Keller, R. M. (1982) A survey of research in strategy acquisition. *Technical Report DCS-TR-115, Computer Science*, Rutgers University, New Brunswick, NJ.

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

Klopf, A. H. (1972) Brain function and adaptive systems—A heterostatic theory. Air Force Cambridge Research Laboratories Research Report, AFCRL-72-0164, Bedford, MA. (A summary appears in *Proceedings International Conference on Systems, Man, Cybernetics*). IEEE Systems, Man, and Cybernetics Society, 1974, Dallas, Texas.

Klopf, A. H. (1982) *The hedonistic neuron: A theory of memory, learning, and intelligence*. Washington, D.C.: Hemisphere.

Klopf, A. H. and Gose, E. (1969) An evolutionary pattern recognition network. *IEEE Transactions on Systems, Science and Cybernetics*, *15*, 247–250.

Korf, R. E. (1985) Macro-Operators: A Weak Method for Learning. *Artificial Intelligence*, *26*, 35–77.

Langley, P. (1982) Language acquisition through error recovery. *Cognition and Brain Theory*, *5*, 513–541.

Langley, P. (1983) Learning search strategies through discrimination. *International Journal of Man-Machine Studies*, *18*, 513–541.

Langley, P. (1985) Learning to search: from weak methods to domain-specific heuristics. *Cognitive Science*, *9*, 217–260.

Luger, G. F. (1976) The use of the state space to record the behavioral effects of subproblems and symmetries in the Tower of Hanoi problem. *International Journal of Man-Machine Studies*, *8*, 441–421.

McClelland, J. L. and Rumelhart, D. E. (1986) *Parallel distributed processing: explorations in the microstructure of cognition, V. 2*. Cambridge, MA: Bradford Books.

McCulloch, W. S. and Pitts, W. H. (1943) A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, *5*, 115–133.

Mendel, J. M. and McLaren, R. W. (1970) Reinforcement learning control and pattern recognition systems. In *Adaptive, Learning and Pattern Recognition Systems: Theory and Applications*, ed. by Mendel, J. M. and Fu, K. S., Academic Press, New York.

Michie, D. and Chambers, R. A. (1968) BOXES: An experiment in adaptive control. *Machine Intelligence 2*, ed. by Dale, E. and Michie, D., Edinburgh: Oliver and Boyd, pp. 137-152.

Minsky, M. L. (1954) Theory of neural-analog reinforcement systems and its application to the brain-model problem. Princeton University Ph. D. Dissertation.

Minsky, M. L. (1963) Steps toward artificial intelligence. *Proceedings of the Institute of Radio Engineers*, 1961, *49*, 8–30. (Reprinted in *Computers and Thought*, ed. by Feigenbaum, E. A., and Feldman, J., New York: McGraw-Hill, 406–450.)

Minsky, M. L. and Papert, S. (1969) *Perceptrons: An introduction to computational geometry*. Cambridge, MA, MIT Press.

Minsky, M. L. and Selfridge, O. G. (1961) Learning in random nets. *Information Theory*, Fourth London Symposium. London: Butterworths.

Mitchell, T. M. (1977) Version spaces: A candidate elimination approach to rule learning. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, MA., pp. 305–310.

Mitchell, T. M. (1982) Generalization as search. *Artificial Intelligence*, *18*, 203–226.

Mitchell, T. M., Utgoff, P. E., and Banerji, R. (1983) Learning by experimentation: acquiring and refining problem-solving heuristics. In *Machine Learning*, ed. by Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., Tioga Press, Palo Alto, CA.

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

Nilsson, N. J. (1965) *Learning machines*. McGraw-Hill, Inc.

Nilsson, N. J. (1971) *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, Inc.

Ohlsson, S. (1982) On the automated learning of problem solving rules. *Proceedings of the Sixth European Meeting on Cybernetics and Systems Research*, 151–157.

Parker, D. B. (1985) Learning-logic. *Center for Computational Research in Economics and Management Science, Technical Report TR-47*, Massachusetts Institute of Technology, Cambridge, MA.

Politis, D. T. and Licata, W. H. (1986) Adaptive decoder for an adaptive learning controller. *Proceedings of SPIE Applications of Artificial Intelligence III*, Orlando, Florida.

Raibert, M. H. (1986) Legged Robots. *Communications of the ACM*, *29*, 499–514.

Rastrigin, L. A. (1963) The convergence of the random search method in the extremal control of a many-parameter system. *Automation and Remote Control*, *24*, 1337–1342.

Reilly, D. L., Cooper, L. N. and Elbaum, C. (1982) A neural model for category learning. *Biological Cybernetics*, *45*, 35–41.

Rendell, L. A. (1983) A new basis for state-space learning systems and a successful implementation. *Artificial Intelligence*, *20*.

Rendell, L. (1985) Substantial constructive induction using layered information compression: tractable feature formation in search. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, pp. 650–658.

Rosenblatt, F. (1962) *Principles of neurodynamics*. New York: Spartan Books.

Rumelhart, D. E. and Zipser, D. (1985) Feature discovery by competitive learning. *Cognitive Science*, *9*, 75–112.

Rumelhart, D. E. and McClelland, J. L. (1986) *Parallel distributed processing: explorations in the microstructure of cognition, V. 1.* Cambridge, MA: Bradford Books.

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

Samuel, A. L. (1959) Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, *3*, 210–229.

Saridis, G. N. (1970) Learning applied to successive approximation algorithms. *IEEE Transactions on Systems Science and Cybernetics*, *SSC-6*, 97–103.

Schaefer, J. F. and Cannon, R. H., Jr. (1966) *On the control of unstable mechanical systems*. International Federation of Automatic Control, London.

Selfridge, O. G. (1959) Pandemonium: A paradigm for learning. *Proceedings of the Symposium on the Mechanisation of Thought Processes*. Teddington, England: National Physical Laboratory, H.M. Stationary Office, London, 2 vols.

Selfridge O., Sutton, R. S. and Barto, A. G. (1985) Training and tracking in robotics. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA.

Smith, S. (1980) A learning system based on genetic algorithms. University of Pittsburgh Ph. D. Dissertation.

Smith, F. G., Mitchell, T. M., Chestek, R. A., and Buchanan, B. G. (1977) A model for learning systems. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 338–343.

Soklic, M. E. (1982) Adaptive model for decision making. *Pattern Recognition*, *15*, 485–493.

Stafford, R. A. (1963) Multi-layer learning networks. In *Symposium on self-organizing systems*, ed. by Garvey, J. E., Office of Naval Research.

Stafford, R. A. (1965) A Learning Network Model. In *Biophysics and Cybernetic Systems*, ed. by Maxfield, M., Callahan, A., Fogel, L. Spartan Books, Inc., Washington, D.C.

Sussman, G. J. (1975) *A computer model of skill acquisition*. New York: American Elsevier.

Sutton, R. S. (1984) Temporal aspects of credit assignment in reinforcement learning. University of Massachusetts Ph. D. Dissertation.

Sutton, R. S. (1985) Personal presentation to PDP research group, University of California, San Diego.

Sutton, R. S. (1986) Two problems with backpropagation and other steepest-descent learning procedures for networks. Poster Session at Eighth Annual Meeting of the Cognitive Science Society, Amherst, MA.

Thorndike, E. L. (1911) *Animal intelligence*. Darien, Conn.: Hafner.

Touretzky, D. S. (1986) BoltzCONS: reconciling connectionism with the recursive nature of stacks and trees. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, Amherst, MA., pp. 522–530.

Touretzky, D. S. and Hinton, G. E. (1985) Symbols among the neurons: details of a connectionist inference architecture. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA.

Uhr, L. and Vossler, C. (1961) A pattern recognition program that generates, evaluates and adjusts its own operators. *Proceedings of the Western Joint Computer Conference*, 555–569.

Utgoff, P. E. (1986) Shift of bias for inductive concept learning. In *Machine Learning: An Artificial Intelligence Approach, Volume II*, ed. by Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., Morgan Kaufmann, Los Altos, CA.

Waterman, D. A. (1970) Generalization learning techniques for automating the learning of heuristics. *Artificial Intelligence*, *1*, 121–170.

Widrow, B. (1962) Generalization and information storage in networks of ADALINE "neurons." In *Self-organizing systems*, ed. by Yovits, M., Jacobi, G., and Goldstein, G., Spartan Books.

Widrow, B. and Smith, F. W. (1964) Pattern-recognizing control systems. *1963 Computer and Information Sciences (COINS) Symposium Proceedings*, Washington, D.C.: Spartan.

Williams, R. (1986) Reinforcement learning in connectionist networks: A mathematical analysis. *Technical Report TR-8605, Institute for Cognitive Science*, University of California, San Diego.

Winston, P. H. (1975) Learning structural descriptions from examples. In *The Psychology of Computer Vision*, ed. by Winston, P. H., McGraw-Hill, New York.

Yudin, D. B. (1966) Quantitative analysis of complex systems II. *Engineering Cybernetics*, *1*, 1-13.