CONTINUAL LEARNING IN REINFORCEMENT ENVIRONMENTS

by

MARK BISHOP RING, A.B., M.S.C.S.

DISSERTATION
Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN
August, 1994

For My Father,
without whom I never would have done this.
And To Amy,
without whom I never could have done this.

And In Memory of
Robert F. Simmons

Acknowledgments

This book is a modestly revised version of my dissertation of the same title which was written for the Department of Computer Sciences at the University of Texas at Austin. The corrections, revisions, and preparations for printing in book format were all done at the GMD — German National Research Center for Information Technology, in the Institute for System Design Technology, where I serve as a postdoc in the Adaptive Systems Research Group, led by Dr. Heinz Mühlenbein.

Work on the dissertation began with an effort to understand human thought processes. The first major step in this journey was discovering that it had already been done over 100 years ago. Through *The Principles of Psychology*, I learned from William James that many of my early ideas were promising, if incomplete, and should be pursued. Though none of this research survived into the final dissertation, the foundation and understanding I gained into human psychology has continually benefited all aspects of my research. I therefore want to acknowledge, first and foremost, my advisor, Robert Simmons. I will always remember him as the person who introduced me to a true theory of human psychology, who waited patiently as my thoughts developed, and who gave me (much) more than 600 hours of his time while I tried to explain them. Robert Simmons was a rare man, a professor who truly cared about his students and made sure that they were able to overcome the many and constant challenges of graduate school. This work would not have been possible without him. I will always owe him a great debt, and I will miss him.

Much of this dissertation was greatly improved by the contributions of David Pierce, who read and commented on all the chapters (in some cases more than once!), and who was a daily help to me during the arduous process of thinking through and writing up these thoughts. Kadir Liano acted as my sounding board before Dave, and to him I owe much gratitude for his patience and for his excellent understanding of mathematics which he always generously shared with me. I would like to thank Rick Froom for his friendship, which has been a constant source of support for me. I've also been very fortunate to have had Eric Hartman's personal and professional support (as well as his sense of humor) to rely upon for many years. Thanks to Jim Keeler for sharing his time and confidence and for showing me what it means to work like hell. I'd also like to thank Ben Kuipers for his help and encouragement, and to him and his students for allowing me the use of their sleeker, faster, more expensive computers for doing many weeks of constant computations. I thank Pavilion Technologies for allowing me to use their equipment while I worked for them.

Special thanks to Peter Dayan, who was more than generous with his time and energy, reading through the entire dissertation and giving me many invaluable comments, particularly on the sections involving reinforcement learning. Thanks also to Risto Miikkulainen and Ray Mooney, whose comments also had a substantial impact on the final version. Long-Ji Lin was very helpful in working through issues related to those described in Section 5.5. Ming Tan pointed out the importance of separating the testing of Continual Learning from that of testing CHILD. Joseph O'Sullivan gave me software for generating 3D policy plots, such as that in Figure 5.3. Many thanks to Jürgen Schmidhuber for his encouragement and for discussions on many topics.

Thanks in a very major way go to NASA who supported three years of this research through their Graduate Student Researchers Program, and to Tim Cleghorn at NASA who has always shown enthusiasm for my work and who encouraged me despite many delays.

The most important acknowledgment belongs to the person whose contribution was greater than all others combined: Amy Graziano (my wife). Without her help, I would now be much older and much thinner.

Abstract

Continual learning is the constant development of complex behaviors with no final end in mind. It is the process of learning ever more complicated skills by building on those skills already developed. In order for learning at one stage of development to serve as the foundation for later learning, a continual-learning agent should learn hierarchically. CHILD, an agent capable of *Continual, Hierarchical, Incremental Learning* and *Development* is proposed, described, tested, and evaluated in this dissertation. CHILD accumulates useful behaviors in reinforcement environments by using the *Temporal Transition Hierarchies* learning algorithm, also derived in the dissertation. This constructive algorithm generates a hierarchical, higher-order neural network that can be used for predicting context-dependent temporal sequences and can learn sequential-task benchmarks more than two orders of magnitude faster than competing neural-network systems. Consequently, CHILD can quickly solve complicated non-Markovian reinforcement-learning tasks and can then transfer its skills to similar but even more complicated tasks, learning these faster still. This continual-learning approach is made possible by the unique properties of Temporal Transition Hierarchies, which allow existing skills to be amended and augmented in precisely the same way that they were constructed in the first place.

Table of Contents

List of Tables

List of Figures

1

Introduction

The real world is characterized by a seemingly unlimited degree of detail and regularity. Regularity occurs at multiple scales and to various extents. The simplest organisms may find regularity such as correlations between scent and food (or movement and danger) that allow them to survive and succeed. More developed creatures recognize subtler concepts, such as the movements involved in mating rituals, and carry out more complex behaviors, such as hunting and stalking prey. Humans, during the course of their lives, continually grasp ever more complicated concepts and exhibit ever more intricate behaviors. The world supports this continual learning process by providing a never-ending multitude of complexities and regularities.

Traditional AI, dedicated to the automation of processes formerly requiring human intelligence, is also characterized by continual development. As the field progresses, new technologies are created from those already in use. Machine learning techniques assist and speed this process by allowing developers to focus on the more abstract issue of what the working system must *do* rather than on how the system should *work*. (This is usually done by creating examples of the system's desired outputs for different inputs.) Developers often find that even the much simpler task of specifying what the system must do can nevertheless be onerous and time-consuming; so the pressure of progress also forces learning algorithms to improve. Reinforcement-learning can potentially speed AI progress by taking the process of abstraction one step further: the developer of a reinforcement-learning system need not even *specify* what the working system *does*, but need only *recognize* when the system *does the right thing*.

This dissertation focuses on an issue still more abstract: the process of automating the development process itself. This is *continual learning*. Given a world of unlimited complexity and regularity, and a method for measuring improvement, learning does not need to stop. Even while performing a skill already well learned, it may be possible for learning to continue, blurring the traditional distinction between training and performance. Like the evolution of an organism, the growth of human learning, and progress in the field of AI, *continual learning* is the process of constant improvement, towards no single, final end other than improvement itself.

1.1 Ingredients of Continual Learning

Constructing an algorithm capable of continual learning is a difficult business. Fields of research can progress because this progress is made by humans — systems still somewhat too complicated to simulate computationally. Humans are themselves only capable of research after decades of a continual-learning process requiring an unfathomably large number of experiences and a device perhaps too complicated for humans ever to fully comprehend. (Some say that if the brain were simple enough to understand, we'd be too simple to understand it.) We can, however, attempt to specify a simplest minimal set of ingredients

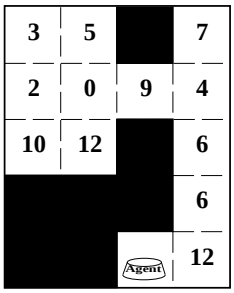

Figure 1.1: A simple Environment.

necessary for the continual-learning process and then try to construct and combine these ingredients computationally.

Three initial ingredients of continual learning are as follows. First, the continual-learning algorithm should be autonomous: it should be able to receive input information, produce outputs that can potentially affect the information it receives, and respond to positive and negative reinforcement. That is, it must *behave* in its environment and be able to assign credit to behaviors that lead to desirable or undesirable consequences. Second, these behaviors should be capable of spanning arbitrary periods of time; i.e, their duration should have no preset limit. Third, the continual-learning algorithm should be able to acquire new behaviors when useful, but should avoid acquiring them otherwise.

1.2 An Example

A simple example should help demonstrate what is required by continual learning. Imagine an agent in a maze-like environment, such as that shown in Figure 1.1. The agent can occupy any of the twelve numbered positions. In each position the agent perceives a number, which uniquely represents the configuration of the walls immediately surrounding that position. (A more detailed description will be given in Section 7.3.) The agent can move north, south, east, or west. (It may not enter the barrier positions in black, nor may it move beyond the borders of the maze.)

The agent's task is to learn to move from any position in this grid-world maze to the goal state (marked by the food dish), where it receives reinforcement. Using standard reinforcement-learning techniques (as will be described in Chapter 5), the agent can learn which actions to take so that it will reach the goal no matter where it begins. Such an agent is autonomous: it receives information, evaluates it, makes a decision, and acts without human intervention or interruption. (The relationship between the agent and its environment will be discussed in Chapter 2.)

All positions are uniquely specified by the number in that position, except for those labeled "6" and "12". However, if in all positions labeled "6", the agent moves south, and in all labeled "12" the agent moves west, then it can choose an action in each position based only upon its immediate perception, that will lead it to the goal regardless of its starting position. But what should happen if the agent is moved to a slightly more complicated environment, such as that shown in Figure 1.2, where there is an ambiguity due to the two occurrences of the input "9"? The agent's previous learning should remain for the most part

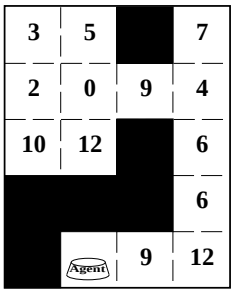

Figure 1.2: A slightly more complex environment.

intact. The agent should continue to move east when it senses "9" in the upper position. However, it should move west upon sensing "9" in the lower position. How can this be done without disturbing the behavior the agent has already successfully learned that brings it to the goal in Figure 1.1?

The answer proposed in this dissertation is to build upon the extant skills hierarchically, leaving what is in place and amending it only as necessary to accomplish the current task. The agent must leave its response to stimulus "9" in the upper position intact while modifying its response to the same stimulus in the lower position. It must therefore be able to distinguish the two different positions, though its input remains the same. It can recognize that it is in the lower position by its preceding context: only in the lower position has it just seen input "12". Therefore, the agent's response to the "9" stimulus must be mediated by the agent's preceding sensory information.

If the agent were now to be placed in a different maze, such as that shown in Figure 1.3, its behavior would need to be extended again, such that it moves west upon seeing "9" whenever its previous input was either a "12" *or* a "9". The agent should cope with every new situation in a similar way: old responses should be modified with surgical precision, and new exceptions to these responses should be implemented based on the contextual information that disambiguates the situations in which they apply.

1.3 Incremental Learning, Hierarchical Development

Two further ingredients of continual learning manifest themselves in the above examples. The first is *incremental learning*. Incremental learning is a continuing process whereby

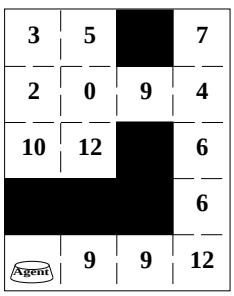

Figure 1.3: A still (slightly) more complex environment.

learning occurs with each experience rather than from a fixed and complete set of data. Many learning algorithms, such as ID3 and back-propagation are so-called *batch* algorithms requiring all training data to be collected in advance of the algorithm's execution. (Back-propagation can be modified trivially, however, into an incremental version that allows data to be generated and presented as training progresses; and incremental versions of ID3 — ID4 and ID5 — also exist.) For continual learning, it is not known in advance what problems will be addressed. It is impossible to collect the data from all the problems in advance of training, and therefore incremental learning is needed.

The second and by far the most significant ingredient of continual learning is *hierarchical development*. Hierarchical development is the subsumption of extant mechanisms or behaviors by newer, more sophisticated ones. This bottom-up process uses the system's old components as constituents of newly created components. Richard Dawkins [18] describes evolution as a similar process whereby complex organisms evolve from simpler ancestors that already embody many of the later organism's sub-assemblies. It seems reasonable to speculate also that elaborate *behaviors* would be exhibited by organisms whose ancestors performed less elaborate versions of these behaviors. Human bipedal locomotion, for example, is a very sophisticated behavior that arose from a less demanding, quadrupedal form of movement. One of the clearest examples of this kind of development is the *subsumption architecture* of Brooks [14] that seems to mimic the evolutionary process described by Dawkins. Though development occurs in the subsumption architecture only through concentrated human effort, it represents precisely the kind of development that must occur automatically in continual learning.

1.4 Existing Hierarchical Systems

The importance of hierarchy in adaptive systems that perform temporal tasks has been noted often, and many hierarchical systems have been proposed. In existing systems such as those of Albus [1], Roitblat [86, 87], Jameson [42], Lin [55], Wixson [126], Dayan and Hinton [21], Schmidhuber and Wahnsiedler [97], and Singh [99], hierarchical architectures are developed top down by hand as an efficient method for modularizing large temporal tasks.

In the architectures of Albus and Roitblat, tasks are defined as a disjunction of sequences of subtasks. That is, every task might be accomplished in several ways, each of which involves executing a sequence of less elaborate tasks. The other systems mentioned above make use of reinforcement-learning. Wixson's, Lin's, and Dayan and Hinton's systems correspond roughly to the Albus-style architecture without the disjunction: Each high-level task is divided into sequences of lower-level tasks where any task at any level may have a termination condition specifying when the task is complete. Jameson's system is somewhat different in that the higher levels "steer" the lower levels by adjusting their goals dynamically. Schmidhuber and Wahnsiedler proposed a mechanism for decomposing a start and goal combination into a fixed number of subgoals generated automatically to minimize the "cost" of the action sequence. The system proposed by Singh, when given a task defined as a specific sequence of subtasks, automatically learns to decompose the task into its constituent sequences.

In all of the above systems, hierarchy is enlisted for task modularization. This allows higher levels to represent elaborate behaviors that span broad periods of time. Modularization can also speed learning. For example, the learning algorithm can take into account the slower time-scales at which higher levels operate when assigning credit to the different levels of the hierarchy. This "vertical" credit assignment is different from but related to both temporal and structural credit assignment, and it is important in any temporal, hierarchical learning system.

The purpose of the above systems, however, is not to develop hierarchies bottom-up as a method for learning more and more complicated tasks. Rather, their purpose is to improve performance in predesignated domains. These systems are constructed top-down by *human* intelligence to reflect the intuitive decomposition of the task and its subtasks (though Wixson presented some general guidelines for determining how to create new hierarchical nodes).

1.4.1 Bottom-up Constructive Hierarchies

There are existing hierarchical systems that *do* develop their architectures bottom-up, such as those of Wilson [124] and Drescher [24]. Wilson proposed a bucket brigade system that allows classifiers to be executed in a hierarchical fashion. The system in some ways resembles that of Albus (above) but its foundation in a classifier system implies the possibility of automatic hierarchy-construction by a genetic algorithm. The schema system proposed by Drescher supports three kinds of dynamic architectural development: new schemas, "composite actions" (sequences of actions that lead to specific goals) and "synthetic items" (concepts used to define the pre-conditions and results of actions). The latter two are hierarchical constructs and reflect the fact that Drescher's goal — simulating early stages of Piagetian development — is most congruous with the philosophy of continual learning. Drescher's is also the most intricate system of those mentioned here.

Macro-operators in STRIPS [7, §D5] and "chunking" in SOAR [52] are two other methods for constructing temporal hierarchies. Macro-operators are specific sequences of lower-level operators combined into a single new operator. They are somewhat like non-disjunctive Albus-style hierarchies, though they are composed at the lowest level of discrete actions, whereas Albus's hierarchies consist of continuous actions. Unlike Albus hierarchies, macro-operators can be constructed automatically to represent frequently occurring sequences. Chunking in SOAR is also a development process. In solving a problem or task, the solutions to subproblems are remembered (as a "chunk") and used again whenever the subproblem reappears. In real-world tasks, both macro-operators and chunking tend to be less suitable than methods based on closed-loop control such as reinforcement-learning. With macro-operators and chunking there is an assumption that the results of actions are known in advance. They are not learned, which makes learning in stochastic environments cumbersome.

Another constructive, bottom-up approach is the "hierarchy of decisions" of Dawkins [18], similar to "history compression," recently described and implemented by Schmidhuber [96]. The idea is that if one element of a sequence reliably predicts the next several elements, then it can represent the predicted elements in a reduced description of the entire sequence. For example, the sequence: `AbcDefXyzDefQrsXyzDefQrsAbcDef` can be reduced to `ADXDQXDQAD`. This new sequence could then be reduced in the same way, to `AXXA`, which represents the

original sequence. This bottom-up process can continue, constructing a many-leveled hierarchy for long sequences with a large amount of regularity. However, this is not an incremental method: all data must be specified in advance. It is also not immediately obvious how to convert a method that constructs hierarchies out of a set of data into something an agent can use for choosing actions.

1.5 Proposed Methods

In this dissertation I introduce a system capable of *Continual, Hierarchical, Incremental Learning* and *Development* (CHILD). Two hierarchical methods are explored for use in CHILD. The first method constructs hierarchies of binary sequences. The elements at the lowest level of the hierarchy can be either senses or actions. Higher-level elements are similar to macro-operators in that they stand for specific sequences of lower-level nodes. These elements are different from macro-operators in that each unit acts as both an action and as a sense: it can be executed, and it generates a value indicating whether the sequence it represents occurred. The units are embedded in a connectionist-like system allowing operation in stochastic environments. An overview of this method is presented in Section 6.1.

The second method, *Temporal Transition Hierarchies*, is far more successful than the first. It is a neural-network-based learning algorithm presented in detail in Section 6.2. It resembles Dawkins' "hierarchy of decisions" method in that it pays particular attention to the least predictable events. It then creates new units that learn to predict these. However, the Temporal Transition Hierarchies method keeps track of the *probabilities* between events (there may be no such thing as a completely "reliable" sequence of events), and it uses the new units to modify these probabilities dynamically.

The underlying assumption of the untrained Temporal Transition Hierarchies network is that event probabilities are constants: "the probability that event A will lead to event B is P_{AB}." The network's task is to learn these probabilities. Such an assumption of constancy is only the first, coarsest, and most abstract description of any set of events, however. After some examination, certain events can be seen to follow other events with *varying* probabilities depending on the context. There may be a specific probability that pressing the right button on the vending machine will result in the sudden appearance of a small box of doughnuts. However, knowing the context — whether the correct amount of change was deposited in the slot — generates two different, much more precise probabilities.[1]

The Temporal Transition Hierarchies algorithm focuses on highly unpredictable events and creates new units to help predict these events more reliably. The new units look at information from the previous time step in search of an unambiguous context in which the event becomes predictable, just as the context of "deposited correct change" helps determine whether "doughnuts will appear" when the right button is pressed.

In robotics tasks, an action may seem to succeed frequently and fail frequently. Temporal Transition Hierarchies can be used to find the broader context in which an action will

[1] Dawkins also noticed this with respect to animal behavior, and he posited a model in which "there is not just one global set of transition rules governing all behaviour patterns of an animal," but that there are "nested sets of transition rules, each set of rules holding sway within a circumscribed cluster of elements." Dawkins' model is also hierarchical, though it is much different from and somewhat less powerful than the one presented here.

succeed. In Figure 1.2 simply sensing input "9" is insufficient for determining whether to move east or west. A new unit would be constructed to find the broader context in which the agent should move east. The new unit searches for information from the preceding time step to predict whether, when the agent senses "9", the move-east action will succeed or fail. If it finds no such information, another unit can be built to search one more step back in time, and so on.

The units created by the Temporal Transition Hierarchy resemble Drescher's "synthetic items", created to determine the causes of an event (the cause is determined through training, after which the item represents that cause). What is particularly powerful about Temporal Transition Hierarchies is that they are differentiable and can be trained via gradient descent, as described in Section 6.2.3. The algorithm can therefore also be used as a sequence-learning neural-network algorithm. As such, it learns very quickly — more than two orders of magnitude faster than recurrent neural-network algorithms on benchmark tasks — shown in Section 7.2.

1.6 Overview of Dissertation

The chapters that follow fall into three categories: background material, novel contributions, and results. Chapters 2–4 are background chapters, providing descriptions of all concepts necessary for understanding the primary technical contribution (Chapter 6), the results (Chapter 7), and the conclusions (Chapter 8). Chapter 2 explores the kinds of tasks that appear in the later chapters and describes their most influential attributes. It is a general overview of the technical issues relevant to continual learning, such as sense and action types, Markov environments, reinforcement versus supervised learning, and modeling versus control. Chapter 3, "Neural Network Learning," describes background issues and related research in the field of neural networks most relevant to continual learning and the methods of Chapter 6. Besides simple feed-forward networks, it discusses constructive and higher-order neural networks. Chapter 4 discusses the issue of time in neural networks, including time-delay neural networks, recurrent neural networks, constructive recurrent neural networks, and higher-order recurrent neural networks. Chapter 5 is a general overview of reinforcement learning and describes the most pertinent aspects of the field for those not already familiar with it. Chapters 3, 4, and 5 may be skipped in part or in whole by those who already have a good background in the topics addressed. Chapter 6 describes the two hierarchical methods mentioned just above. It derives the learning rule for the second of these (Temporal Transition Hierarchies) and presents the learning algorithm. Chapter 7 presents the major results, first demonstrating the efficacy of the Temporal Transition Hierarchies learning algorithm, and then presenting CHILD, the continual learner, in reinforcement-learning environments. Chapter 8 concludes the dissertation by discussing and interpreting the results, and analyzing the contributions as well as the deficiencies of the Temporal Transition Hierarchies algorithm and of CHILD as a continual learner. The chapter ends optimistically by proposing future work.

2

Robotics Environments and Learning Tasks

This chapter explores the kinds of tasks that appear in the chapters that follow. These tasks can best be described as simple (often simplistic) robotics tasks that have several important properties. First, they are amenable to reinforcement learning, where the reinforcement can be easily changed to create tasks of greater or lesser difficulty. Second, they are easy to visualize: it is clear what the robot (or *agent*) should be learning, and its progress can be readily measured by monitoring the amount of reinforcement it receives. Third, the tasks are highly general; the details, including the complexity of the task to be learned, are all modifiable. In fact, all tasks of this kind can be described very simply in terms of senses, actions, and reinforcement, which allows an enormous range of possible specific tasks, from the trivial to the non-computable.

2.1 Robots and Robotics Tasks

Any robot can be described as the implementation of a set of mappings from current and previous sense signals and actions, to action signals. For a robot to perform a task it must produce actions through its actuators, possibly as a function of its previous actions and its previous and current sensations. In the discrete time case, this can be formalized as follows:

$$\vec{a}(t) = f_t(\vec{s}(0), \vec{a}(0), \vec{s}(1), \vec{a}(1), \ldots, \vec{s}(t-1), \vec{a}(t-1), \vec{s}(t)), \tag{2.1}$$

where $\vec{a}(\tau)$ is a vector of actuator signals describing the motor activity of the robot at time τ; $\vec{s}(\tau)$ is the vector of sensory signals the robot receives at time τ; and f_t is a function mapping a sequence of $2t+1$ vectors onto a single vector. f_t is not necessarily deterministic but might choose randomly from among many possible action-vector candidates. This formalization is general enough to describe any discrete-time robot.[1] The senses can encode tactile, auditory, visual information, etc. including, for example, joint angles and rates of change. The action vector can encode any control signal, including the specification of positions, joint angles, and torques.

Since f_t can be *any* function taking the proper arguments, Equation 2.1 imposes no limits on the complexity of the robot. A more appealing yet no less general formulation of Equation 2.1 is the following combination of equations:

$$\vec{a}(t) = f(S(t)) \tag{2.2}$$

$$S(t) = g(S(t-1), \vec{a}(t-1), \vec{s}(t)). \tag{2.3}$$

Clearly, Equations 2.2 and 2.3 are identical to Equation 2.1 when g is the concatenation operator, and $f(S(t))$ simply translates its argument into a call of f_t. However, it is convenient to think of the robot's next action as a function of its last action, its current sensory inputs, and its internal state.

[1] The continuous-time case will not be considered here.

2.1.1 The Environment

The robot's *environment* interprets the sequence of action vectors and generates the sequence of sense vectors. It can be described as nearly the mirror-image of the robot:

$$\vec{s}(t) = f'(E(t)) \tag{2.4}$$
$$E(t) = g'(E(t-1), \vec{a}(t-1)). \tag{2.5}$$

where $E(t)$ is the state of the environment at time t. (Just as with f in Equation 2.1, both f' and g' may be stochastic: different possible states might result from a given action in a given state, and different possible sense vectors can be produced in the same state on different occasions.) A similar description of finite-state task environments was given by Wilson [125].

2.1.2 The Task

Together, Equations 2.2–2.5 define a protocol by which a robot can interact with an environment. The robot acts in response to the sensations it receives; the environment responds to the robot's actions. This general framework describes a set of robots that can perform the broadest range of tasks in the broadest range of environments.

Besides describing the actions that are performed by a robot, the functions f and g implicitly describe the *task* the robot performs. Since Equations 2.2–2.3 impose no limits on the complexity of the robot, they therefore also impose no limit on the complexity of the robot's task (provided it can be performed at all). However, the task might often require *less* than all the information supplied in Equation 2.1. Some very simple tasks do not depend on sensory or action information of any kind. In fact, most tasks, even most of those that require knowledge of previous actions, can be performed when g is a function of state and current sense information only (i.e., when f_t is a function of sense information only). With proprioceptive devices, for example, the robot can encode its last action as sensory inputs. This simply uses the robot's hardware to transform action information into sensory information, thereby eliminating g's explicit dependence on $\vec{a}(t-1)$. More generally, g could include as part of its preliminary computation of $S(t)$ a computation of $f(S(t-1))$. This will work when f is deterministic or when its randomness is reproducible. The function g actually *requires* $\vec{a}(t-1)$ as an argument only when (1) the task requires knowledge of previous actions, (2) f is truly stochastic, and (3) the robot lacks sufficient proprioceptive devices.

2.1.3 The Learning Task

Because $\vec{a}(t)$ can be used to describe the behavior of an agent that performs a task, it can also be used to express the desired behavior of an agent that *learns* to perform the task. In this case, Equations 2.2 and 2.3 describe a set of training examples for a supervised-learning agent. The training input to the agent at time τ would be $\vec{s}(\tau)$ — and possibly $\vec{a}(\tau-1)$ — and the target output would be $\vec{a}(\tau)$.

Reinforcement Learning Tasks. A *reinforcement-learning* agent is somewhat more sophisticated than the supervised-learning agent. A teacher must be present to provide the supervised-learning agent with correct responses for each situation. In reinforcement learning, the correct action is never given. Instead, the agent must learn for itself which actions are correct in each situation.

To make learning possible without a teacher, a reinforcement environment supplies the agent with a "reinforcement signal." The agent monitors changes in the reinforcement signal to decide which actions are *best*, where the best actions maximize the agent's expected reinforcement over time. More formally, the reinforcement signal is some function of the previous state of the environment and the most recent action taken:

$$r(t) = R(E(t-1), \vec{a}(t-1)), \tag{2.6}$$

where $E(t)$ was given in Equation 2.5. The *correct* action to take in a state is any action that maximizes the expected future reinforcement:

$$\vec{a}(t) = \operatorname*{argmax}_{\vec{a}} E[\sum_{\tau=1}^{\infty} \gamma^{\tau} r(t+\tau)], \tag{2.7}$$

where $\operatorname{argmax}_a(f(a))$ returns the argument, a, that maximizes $f(a)$; and γ is a "discount factor" — a value often chosen less than 1.0 to avoid infinite sums. The correct action at time t is any of the possible actions that maximizes the expected sum of the (possibly discounted) future reward signals — assuming every action taken at every step obeys this rule.

Because reinforcement-learning environments specify the agent's reinforcements, they also implicitly define the agent's task (to perform the behavior that maximizes expected reinforcement). As a result, there are two separate meanings of the word "task" in reinforcement learning: the behavior the agent should ultimately perform, and the task of learning to perform this behavior. In the remainder of this dissertation the word "task" is intended to denote the latter, the task of learning to perform the appropriate behavior.

Much has been written about reinforcement-learning in both deterministic and stochastic environments [9, 10, 11, 111, 120]. Some of this work will be described in detail in Chapter 5.

2.1.4 Task Environments

In the reinforcement-learning literature, task environments are frequently quite simple and are typically not intended to replicate *actual* environments. Instead, they are used to test out aspects of intelligence a robot might require in a real environment. The tasks explored are therefore often "toy" domains (for example, the maze tasks of Sutton [105]). Yet they are subtly different from most "toy" domains of traditional Artificial Intelligence (e.g., the blocks world of Winograd [7, §F4]). One way of describing this difference is that in the latter case, algorithms are often devised to solve problems that embody some important aspect of reality, whereas in the former case, problems are usually devised to test algorithms that embody some important aspect of intelligence. The underlying motivation for traditional AI systems is also often to emulate some important aspect of intelligence, but not necessarily in a way that can be tested in arbitrary situations. The algorithms investigated here, however, are not designed for the peculiarities of any specific task. Indeed,

the algorithms can be completely separated from the task and quite nicely plugged into any other task that can be expressed in the form of Equations 2.2–2.6. These equations describe an interface protocol between robots and environments (i.e., between learning agents and the tasks they are to learn). Any agent that follows this protocol may be tested in any environment that also follows it. The interface is very general and applies to any reinforcement-learning task.[2]

Presumably, the agent implements some quality of intelligence. This quality can be tested in different situations by placing the agent in different kinds of environments. The agent's performance might vary greatly across these different environments, and may, in fact, be completely miserable in some. Nevertheless, the agent need not be designed for a certain task or even for a certain kind of environment in order to be tested on it. As a result, the simplicity of many environments used in the reinforcement-learning literature is designed intentionally so as to focus on a particular contribution of the learning algorithm and to measure the algorithm's performance with respect to that contribution.

2.2 Environmental Complexities

The environment can be complex in many different ways, some of which are discussed next. It is important to note, however, that a task may be simple even though it takes place in a complex environment. Therefore, when discussing the difficulties introduced by a complex environment, the robot's task is assumed to be a worst-case task: i.e., it is sufficiently demanding that the robot must resolve the most complex problems the environment can introduce in order to choose the correct actions.

2.2.1 Varieties of Senses and Actions

The range of possible senses and actions allowed by Equations 2.2 and 2.3 is enormous, from the simplest to the most advanced, to the futuristic. In the simplest reinforcement-learning environments, sense vectors explicitly represent the robot's exact location, and action vectors produce only a simple set of actions. Sense and action vectors are generally binary in these cases and are encoded locally. That is, in each vector, exactly one item (corresponding to the current state) has a value of 1, and all others have a value of 0. These tasks are helpful for illuminating certain reinforcement-learning algorithms that enlist the mathematics of dynamic programming, as will be described in Section 5.3. Agents only need to learn the optimal mapping from the immediate sensory input (which unambiguously specifies the environmental state) to the best action(s) for that state. (This mapping is known as the optimal *policy*.) Since these vectors are orthogonal, they are conveniently amenable to learning with even the simplest neural networks.

There are many examples of more complicated environments, however, that use distributed senses [3, 54]. These environments are necessary for testing algorithms that can learn complicated policies (i.e., complicated sense→action mappings). Even more difficult (and more realistic) are environments with continuous-valued, distributed sense and action vectors [3, 5, 72, 92, 107]. Tasks in these environments can be difficult not only because of the possible complexity of the policies they might require, but also because the action space is infinite, meaning it is no longer possible to try all actions exhaustively.

[2]An example system showing the generality of this modularity is described in Section 7.1.

2.2.2 Markov Environments

Though mappings from the current sense vector to the correct action vector can be arbitrarily complex, there are also other dimensions of difficulty in robotics tasks. For example, there is the problem of sense-disambiguation. An algorithm that can learn perfectly an arbitrary mapping from the current sense vector to any action vector is not a guarantee of success, since the best action might also depend on previous sensory inputs.

The simplest environments used to study reinforcement learning are Markovian and do not have the problem of ambiguous sensory information. A *Markov environment* is one in which the sequence of states that the agent visits are Markov chains. Briefly, a Markov chain can be described as a sequence of discrete random variables $\boldsymbol{x}(\tau)$ drawn from a set $\boldsymbol{S}$ representing the state space, and

$$P(\boldsymbol{x}(t) = s \,|\, \boldsymbol{x}(0), \ldots, \boldsymbol{x}(t-1)) = P(\boldsymbol{x}(t) = s \,|\, \boldsymbol{x}(t-1))$$

where $t \geq 1$ and $s \in \boldsymbol{S}$ (see Grimmett and Stirzacker [37, Ch. 6], and Papoulis [70, §15-3]). The probability of encountering a given state at one point in the chain depends only upon the last state encountered; the sequences are history independent. Knowledge of previous items in the chain introduces no further information regarding the probability of the next state.

A Markov environment[3] could therefore be described as an environment in which

$$P[E(t) = s \,|\, (E(0), \vec{a}(0)), \ldots, (E(t-1), \vec{a}(t-1))] = P[E(t) = s \,|\, (E(t-1), \vec{a}(t-1))],$$

where, as before, $\vec{a}(\tau)$ is the action the robot takes at time τ, and $E(\tau)$ is the state of the environment at time τ. Assuming a unique sensory input for each environmental state, the task described by a Markov environment in terms of Equation 2.1 is therefore

$$\vec{a}(t) = f(\vec{s}(t)), \tag{2.8}$$

since the correctness of the action chosen at time t can be determined by the sensory input at time t, inasmuch as it can be determined at all.

2.2.3 Markov-k Environments

One might at first wonder whether the characteristics of a Markov environment are shared by an environment in which the probabilities of the next state depend not just upon the current state and action, but, say, upon the past k state/action pairs. A chain of such state sequences could be described more formally as

$$P(\boldsymbol{x}(t) = s \,|\, \boldsymbol{x}(0), \ldots, \boldsymbol{x}(t-1)) = P(\boldsymbol{x}(t) = s \,|\, \boldsymbol{x}(t-k), \ldots, \boldsymbol{x}(t-1)). \tag{2.9}$$

A Markov-k sequence such as this, however, is easily converted to a k-dimensional Markov-1 sequence [70, p. 530] and therefore has the properties of a Markov-1 sequence. (The term *non-Markovian*, however, generally means "not Markov-1.") A robot that is to negotiate a Markov-k environment will need to store the past k sensory vectors in order to take the

[3] Markov environments are also known as *Markov Decision Tasks* (MDT's) and *Markov Decision Process* (MDP's).

correct actions. Provided that knowledge of k is known for a given environment, the task of the learning agent is identical in the Markov-k and the Markov-1 environments. The learning task is thus described as:

$$\vec{a}(t) = f(\vec{s}(t-k+1), \vec{s}(t-k+2), \ldots, \vec{s}(t)).$$

However, if k is *not* known, then the robot must learn this as well. Methods for dealing with this problem are described in Section 4.2.

2.2.4 Finite State Environments

In the real world, contingencies can span unlimited periods of time. If the robot breaks a vase at one time step, the vase will still be broken forever after that. If a robot that remembers only its past k senses breaks a vase and then leaves the room for $k+1$ time steps, upon its return, it cannot accurately predict its perception of the broken vase. Clearly, the robot needs memory that will store information for arbitrary periods of time, and it must be able to use that memory to generate its actions.

A more general task than that of learning a Markov-k environment is that of learning a *finite-state environment*. As with a Markov-k environment, a finite-state environment requires only a fixed amount of information in order to choose the next action. However, in the case of the finite-state environment, more may be required than simply a fixed-sized history of the past several sensory inputs. Instead, some information may be needed from the arbitrarily distant past. These environments correspond to Finite State Automata (FSA).[4]

In finite-state environments the current state can be determined if the starting state is known and a record is kept of all previous actions (state transitions in the FSA). If all loops are always removed from the record (where a loop is a sequence of transitions that lead from some state back to itself), then the record will never need to be longer than the number of unique states in the environment. When the loops are removed, the record may contain information from arbitrarily long ago, unlike in the Markov-k environment where a record of just the *latest* k inputs is sufficient for determining the current state.

2.2.5 Still More Complex Environments

Keeping a record to determine the current state only works for deterministic finite-state environments, in which the transition taken from a state must lead to a single, specific next state. A *stochastic finite-state environment*, however, does not obey this property. The stochastic finite-state environment corresponds to the non-deterministic finite-state automata in which a transition taken from a state can result in arrival at different states in different instances. These environments are a superset of deterministic finite-state environments. They are also very difficult to negotiate, since one can always determine the current state of a finite-state environment given a complete model of the environment and a record of previous state transitions, but this is not the case with stochastic finite-state environments.

[4]The states of a finite-state environment correspond to the states of the FSA. The sensory information emitted by the finite-state environment corresponds to the "outputs" of states in the FSA. The actions taken by the robot correspond to the transitions of the FSA.

Even more difficult than stochastic finite-state environments are *hidden Markov environments*, in which an underlying stochastic finite-state automaton (i.e., a Markov chain) is not observable directly but can only be observed through a *stochastic* process that generates the observable phenomena from the actual states [78]. Even if the robot has a correct model of the underlying Markov environment and perceptual mappings, it still can only estimate the *probability* that it occupies a particular state; it rarely knows precisely where it is, maintaining at best a fuzzy concept of its state.

As difficult as hidden Markov environments might be, further difficulties can be added. As with finite-state and stochastic finite-state environments, the less knowledge the robot has, the more difficult tasks in that environment may be. If the robot has no knowledge of the observation probabilities, the transition probabilities, or the number of states in the environment, these must also be learned, resulting in the possibility for some excruciatingly difficult tasks. Nevertheless, even more difficult environments could be created if the underlying model were not finite state. Push-down environments, for example, could require arbitrary amounts of information to be stored to solve a given task. These then might be either deterministic or stochastic, etc.

2.2.6 The Effect of Future Activity

Besides the complexities just discussed, there is the issue of reinforcement, which introduces the dimension of *future* activity in choosing correct actions. These aspects of complexity have also been discussed by Littman [58], building on the work of Wilson [125]. Littman describes two dimensions: (1) the number of future steps explicitly or implicitly considered before taking an action (which he labeled β), and (2) the amount of history information needed to take the correct action (which he labeled h).

The most interesting values of β are $\beta = 0$ and $\beta > 0$. In the former case, the only reinforcement of interest is that received immediately upon taking the next action. In the latter case, the agent must choose actions now in order to achieve reinforcements in the (possibly distant) future. Besides this major distinction, it is also of some value to characterize specific tasks by the size of β (i.e., the maximum distance between a reinforcement and a state whose best action is affected by that reinforcement). Clearly, the difficulty of the task increases with β.

Littman assumed a finite-state environment and therefore assumed a finite number of bits, h, could be used to store all the history information needed to achieve maximum reward,[5] since only a finite number of labels are needed to uniquely encode all possible states. But this is not the case in environments that are not finite state, such as push-down environments and most real-world tasks. Real-world events are only reproducible in contrived situations, whereas finite-state environments do not change from one iteration to the next, and events are reproducible: in a given state, the same action will result in the same next states with the same probabilities. In non finite-state environments, instead of states, there are regularities: situations that are similar to previous situations in certain ways.

In non-finite-state environments, one can encounter interesting relationships between h and reinforcement: the maximum achievable reward might be related to the amount

[5]Maximum reward is defined in terms of average reward per time step.

	Dimensions of Complexity	Extreme values
1	Sense/Action Representation	Local vs. Distributed
2	Individual Sense/Action Values	Binary vs. Continuous
3	Sense→Action Mapping	Orthogonal, Linearly-separable, ...
4	Sense→State Mapping	One-one vs. Many-many
5	State→Action Mapping	One-one vs. Many-many
6	Next State Function i.e., (state, action)→state	Deterministic vs. Stochastic Many-one vs. many-many
7	Underlying Model	Markov, F.S.A., P.D.A., ...
8	History Information Needed	$0 \ldots \infty$
9	Duration History Must Be Kept	Fixed vs. Infinite
10	State/Action→Reinforcement Mapping	Many-one vs. Many-many
11	Planning Steps for Reinforcement	$0 \ldots \infty$

Table 2.1: These eleven dimensions of complexity can be used to construct tasks of various kinds of difficulty. The middle column gives the name of the dimension of complexity. The right-hand column gives the extreme cases (when possible) from simplest to most difficult for that dimension. It is conceivable that a particular task may be extremely difficult in one dimension while being trivial in others.

of information the robot can store. The larger the robot's h, the better it becomes at achieving its reward. In these frequent real-world situations (say, taking an exam), reward is achievable with a small h, but is greater with a larger h. Similarly, reinforcement may also be related to any of the other dimensions of difficulty described above.

2.3 Dimensions of Difficulty

The result of the preceding analysis is to demonstrate that there are many different dimensions of complexity present in the robotics tasks of the form given in Equations 2.2–2.6. Some of these dimensions, given in Table 2.1, are as follows. (1) Sense and action vectors can be distributed or encoded locally, and (2) they can have binary, discrete, or continuous values. (3) The mapping from senses to actions may be of any complexity. (4) Senses may unambiguously represent a state or can be highly ambiguous. (5) In the simplest case, there might be a unique action for each state, or in the most complicated case, there are multiple best actions for each state where some states have the same best actions. (6) An action in a state may or may not completely determine the next state. (7) The environment may be a Markov, finite-state, or push-down environment, or something still more difficult. (8) In order to choose the best action, none, some, or all previous sense information may be required. (9) The length of time any particular piece of history information is needed might be fixed (as in Markov-k environments) or arbitrarily long, as in finite-state environments. (10) An action in a state may or may not completely determine the reinforcement to be received. (11) The best action might depend on the current reinforcement alone, or it may depend on future reinforcements as well (Littman's β dimension).

As noted earlier, the complexity of the environment is determined by the most difficult tasks that can be designed for the environment. This in turn implies a required minimal

degree of sophistication on the part of the agent that performs the task. (If a simple agent can perform the task, then it's a simple task.) In general, the greater the skills of the agent along the above dimensions, the more successful it will be in achieving reward in environments that contain complexities across these dimensions. An agent capable of achieving reward in such an environment may achieve greater rewards if its abilities across the dimensions are more acute. If there is no maximum achievable reinforcement, then continual improvement leads indefinitely toward greater average reward.

Modeling versus Control. A word should be said about the distinction between modeling the environment and controlling an agent within it. Many systems, some of which will be described in Sections 5.3 and 5.4, separate these two aspects of learning. Modeling the environment consists of predicting what will happen next if a certain action is taken (including taking no action). The difficulty of the modeling task depends upon dimensions 1, 2, 4, 6, 7, 8, 9, and 10. The control task, on the other hand, is the task of deciding *which action is best*. The difficulty of this task depends upon dimensions 3, 5, 6, and 11. It is convenient to separate control from modeling for several reasons. First, if a controller has a perfect model, it can choose optimal actions by using the model to search the action space [108]. Second, once a model of an environment has been learned, it can be used by many different controllers. Third, the controller may assume as part of its strategy the burden of improving the predictive power of the model by exploring parts of the environment in which the model performs poorly [64, 71, 94, 107]. Finally, learning in both the model and the controller can occur independently but simultaneously, such that incremental advances in the model can increase the efficacy of the controller and vice-versa.

Since different kinds of learning might be required in the modeling and control tasks, certain learning strategies might be more useful in one than in the other. In much of the research on reinforcement learning, some form of neural network is used for one or both tasks. Part of this is due to the generality of networks and their capabilities of learning across so many of the dimensions in Table 2.1, and part of it is due to the way in which search can be done in neural networks by gradient descent. The next chapter is therefore devoted to a description of neural-network learning.

2.4 Conclusions

In the chapters that follow, different methods will be discussed for addressing the dimensions just described. No method (including that proposed in this dissertation) addresses all dimensions perfectly. For continual learning, one would like an algorithm whose potential learning ability is as broad as possible across all dimensions, but which can begin by learning simple tasks and steadily increase its abilities. Think of Table 2.1 as a wish-list that should eventually be addressed by a single learning method. In this dissertation, I discuss a method that does in fact do continual learning, but at the cost of limiting the agent in terms of the complexities of the tasks that it can learn. In particular, CHILD, the agent I will describe beginning in Chapter 6 is limited to learning Markov-k environments. Improvements discussed in the future work section (Section 8.4) attempt to broaden these limitations.

3

Neural-Network Learning

"Neural Network" is a broad term covering a large interdisciplinary field. In this chapter I intend to describe only a large enough part so that the dissertation can be understood. I will first discuss simple feed-forward systems and some of their limitations. Then I will present a few *constructive* algorithms, which build networks as they learn. Finally, I will offer a brief discussion of *higher-order* networks.

3.1 Supervised Mappings

In the previous chapter I discussed the importance in a robotics environment of learning mappings: mappings from senses to states, states to senses, states to actions, etc. An excellent way to encode and to learn these mappings is with a neural network. Standard feed-forward neural networks are capable of representing (if not learning) any computable function mapping [40]. They are not limited to binary- or even discrete-valued inputs or outputs, or to locally encoded pattern representations. This greatly reduces the amount of research effort needed to attack the dimensions of difficulty listed in Table 2.1.

The task faced by neural networks is that of learning supervised mappings: given a training set of input vectors and associated target vectors, learn a *rule* that captures the underlying functional relationship from input vectors to target vectors. That is, each target vector, $\vec{T}_p$, is a function, m, of the input vector, $\vec{I}_p$:

$$\vec{T}_p = m(\vec{I}_p).$$

The task of the network is to learn the function m. This can be achieved by finding regularities in the input patterns that correspond to regularities in the target patterns. The network has at its disposal a set of parameters (weights) whose values can be changed to modify the function m' computed by the network. The parameters are then modified until m' closely resembles m, as measured by its responses to the input patterns of the training set. The network's task is not just to store the training patterns for later retrieval, it is to learn the function m. Learning the *function* allows the network to *generalize* what it has learned to unseen inputs, and to ignore noisy training patterns (input vectors with incorrect targets).

In robotic's tasks, a neural network could be used, for example, to learn the function f in Equation 2.8, where it would take as input the current sense vector and have as a target the desired action vector. This would be effective in Markov-1 environments.

By now, the mechanics of simple feed-forward neural networks and their gradient-descent learning algorithms are quite well known — but a brief description introducing terms and standardizing notation can't hurt. To learn a mapping from a vector of real-valued inputs to a vector of real-valued outputs, each element in the input vector is assigned to a unit (or *neuron*) in the *input layer* of the network. Each element of the output vector is assigned to

a unit in the *output layer* of the network. Typically there is one other layer of units called the *hidden layer*, though there may be any number of hidden layers (including zero), and each may have any number of units. In the typical single hidden layer scenario, each input unit is connected to all the hidden units via *weighted* connections, where the weights are adjusted by the learning algorithm. Each hidden unit is in turn connected to all the output units via a different set of connections. The values of the hidden units are computed as:

$$H^i = f^i(B^i + \sum_j w_{ij} I^j), \tag{3.1}$$

where H^i is the i^{th} hidden unit, B^i is a real-valued bias unit that serves as a threshold, I^j is the j^{th} input unit, w_{ij} is the *weight* of the connection from unit j to unit i, and f^i is an *activation* or *transfer* function (usually a sigmoid such as *tanh* or the logistic function, which are monotonically increasing and have high and low asymptotes with slope zero). The outputs are computed in a nearly identical way:

$$O^i = f^i(B^i + \sum_j w_{ij} H^j), \tag{3.2}$$

where O^i is the i^{th} output unit. Learning is done by comparing the output values to the values they should have been (the *targets*), and using gradient descent to reduce the sum squared difference, E:

$$E = \frac{1}{2} \sum_p \sum_i (T^i - O^i)^2,$$

where T^i is the i^{th} target value and p is an index of input- and target-pattern pairs that are used for training. The weights are the parameters that can be modified by the learning algorithm to reduce the total error generated over all the patterns.

Back-propagation, a gradient descent algorithm, computes the contribution of every weight to the total error: $\Delta w_{ij} \stackrel{def}{=} \frac{\partial E}{\partial w_{ij}}$. The weight is changed in the direction opposite to that contribution: $w_{ij} \leftarrow w_{ij} - \eta \Delta w_{ij}$, where η, the *learning rate*, is usually a small fraction that keeps the weight change from being too large. Gradient descent is a kind of constraint-satisfaction technique: given one set of inputs and another set of outputs, the weights are constrained such that they generate the appropriate output for each input. Teaching a robot the correct action given a particular sense vector (or any of the other mapping tasks described above) is simply a matter of presenting input patterns and target patterns to the network time and time again until gradient descent finds weights that satisfy the constraints of the data.

I stated above that neural networks can in principle represent any function that maps inputs to outputs; however, this applies only to networks with hidden units, and an arbitrarily large number of them at that. Furthermore, the hidden units must have non-linear activation functions (Equation 3.1); otherwise the entire network is no more powerful than a network with no hidden units, and networks with no hidden units have limited representational ability (i.e., they can only make linearly separable classifications). Nevertheless, single-layer systems do have one advantage over networks with hidden units: if the patterns that it must learn *are* linearly separable, the network can learn them very quickly using

gradient descent in the form of the delta rule [118], or, even more quickly using second-derivative information (e.g., Conjugate Gradient [77, §10.6], Quickprop [26]).

On the other hand, more powerful activation functions can increase the abilities of the network, even in the absence of hidden units. As an extreme case, an activation function could be *any* Turing computable function. If the input to the function is in a form that allows identification of the individual input values,[1] a single node can theoretically make any classification.

Less extreme cases are *higher-order* neurons (also called sigma-pi units [88]), which use multiplicative connections — the input is a sum over products of a single weight and any number of input units. This is explained in more detail in Section 3.3. For now it should only be pointed out that even units with a simple, sigmoid activation function can be very powerful if the input is sufficiently sophisticated: units with higher-order inputs of order k (where the sum contains products of k inputs), can solve problems of order k. A problem of order k is one requiring a boolean computation of k variables [62]. (Classification of linearly separable, binary input patterns is a problem of order one.) This is important because Temporal Transition Hierarchies (to be introduced in Section 6.2) contain no hidden units, but do have higher-order units.

3.2 Constructive Networks

A different limitation of neural networks is their fixed architecture. Many algorithms have been proposed that, like the methods to be introduced in Chapter 6, add new units during learning. In general, they are not intended to address continual-learning issues, but rather to address issues of completeness, efficiency, and generalization. If a problem is given to a specific multi-layer network, the network may be too small to solve the problem (i.e., to learn the mapping from inputs to outputs), thus requiring more units before the network can map the entire training set *completely*. On the other hand, large networks are *capable* of learning simple mappings but are inefficient, and their excess parameters usually result in poor generalization. In these cases a small network would be more appropriate. To address these issues, many *constructive* neural networks have been devised that increase the size of the network during the course of learning. I will mention a few.

One of the first network-modifying systems, the Upstart algorithm, was proposed by Frean [31] who suggested adding new hidden units during the course of learning. The network begins as a single perceptron that attempts to learn the complete mapping, which must be a binary classification. Eventually, the learning rule (the "pocket algorithm" [32]) will find the best mapping representable by a single perceptron. At that point there may

[1] This could be done by assigning incoming weights such that the sum in Equations 3.1 and 3.2 can be separated into the original components. For example, if the input values are binary, then weights with values 2^j (where j is the index of the input unit) will generate a sum: $\sum_j^n 2^j I^j$. Since each I^j value will take exactly one bit, the value of I^j can be retrieved as the floor of the sum divided by 2^j modulo 2, i.e. $I^j = \text{floor}(\frac{\sum_j^n 2^j I^j}{2^j}) \bmod 2$.

Alternatively, the notion of an "activation function applied to the sum of the weighted input" can simply be replaced by any function of n parameters, where n is the number of inputs into the neuron.

remain some misclassified patterns. These patterns are broken into two groups, those that were classified incorrectly by the *parent* perceptron as "on" (belonging to the category) or "off" (not belonging to the category). These groups are called the wrongly-on and wrongly-off groups. Two new perceptrons are then created, one to learn the wrongly-on group, and one to learn the wrongly-off group. Strong connection weights are then built from these *daughter* units to the parent to override the parent's misclassification on these patterns. The network's performance will always improve whenever new daughter units are added, since the daughters can *always* improve the output of the parent (i.e., they can simply memorize a single pattern from their training sets and ignore all others, reducing the parent's misclassifications by two — one for each daughter). Eventually the number of classification errors will be brought to zero, resulting in complete learning of the training set.

GAL (for "Grow and Learn" [2]) is a different constructive technique. This method is similar to ART [38] in that it creates each new hidden node to match a specific training pattern. When a pattern is given to the system, each hidden unit is activated in proportion to its Euclidean distance from the pattern that it was created to match. The most highly activated unit "wins" and activates the output units to which it has non-zero connections. When an input pattern activates an output unit that is not in the target pattern, a new unit is then created to match that pattern. The output connections of a new unit are set to 1.0 for all output units that are "on" in the current target pattern, and are set to 0.0 for all output units that are "off" in the current target pattern. GAL also allows connection weights to be modified in a manner similar to that of radial-basis-function networks [63]. When a pattern activates the output units correctly, the hidden unit that "won" is modified so that it will become even more strongly activated the next time this pattern is presented. (This is done by modifying the hidden unit so that the pattern it best responds to is closer in Euclidean distance to the current pattern.) GAL is extremely fast and generally learns a training set in a few passes. Even the *two-spiral problem*, a well-known and very demanding benchmark only requires two passes.

Both the Upstart algorithm and GAL will learn the complete training set, but generalization is to some degree sacrificed, due to the ease with which new units can be created solely for purposes of memorizing a single pattern. Another drawback of GAL and also of the Frean network is that they only work for binary classifications. If the output is continuous, or even if it is discrete but not binary, then the algorithm fails. A related approach that *does* work in the case of non-binary outputs is the Cascade Correlation algorithm [28]. In this algorithm, training is done for just the output units at first, until the network error is no longer decreasing quickly. (The training algorithm is Quickprop, which does gradient descent using second-derivative information.) Once this apparent local minimum has been reached, a pool of *candidate* units is trained to predict the error of the output units. Eventually, training for these candidates also reaches a minimum. At that point, the candidate best at predicting the error is incorporated into the network as a hidden unit and its input weights are frozen. The output units are then re-trained using this new hidden unit to help their prediction. After this has been done, a new pool of candidates are trained, this time taking their input not just from the input units but from the hidden unit(s) as well. This process continues until the error of the network is low enough that the (human) trainer is

satisfied. Because the new units are trained to predict the actual real-valued error of the output units, the algorithm does not require binary targets.

The Upstart algorithm, GAL, and Cascade Correlation, are somewhat inefficient: they create new units with random initial weights and simply allow those units to learn appropriate values to reduce error. A different approach introduced by Wynn-Jones [127] is to configure new units to solve specific problems that the network has encountered while trying to learn the mapping. Wynn-Jones' "Node Splitting" algorithm focuses its attention on those units whose weights during training are being pulled in conflicting directions. That is, the learning algorithm sometimes increases and sometimes decreases the weights: some weights are changed strongly in one direction for some patterns and strongly in the opposite direction for others (while other input weights to the same neuron are perhaps only modified slightly). Node splitting monitors the changes that the learning algorithm calculates for the weights and determines the overall *direction* of the conflict in n dimensions, where n is the fan-in of (i.e, the number of inputs to) the neuron whose weights are in conflict. Two new units are then created to replace the old one. Each new unit's input weights are then assigned to one of the two regions in weight space where the weights of the original unit were being pulled. Training then continues. If the two units were properly placed, their introduction causes very little initial disturbance to the network, and the network continues to train. Of course, it's still possible that *these* nodes may need to be split as well. As will be seen in Section 6.2.4, Temporal Transition Hierarchies use a similar method for deciding when to create new units, though only in one dimension rather than in n.

3.3 Higher-Order Systems

Another way of increasing the power of a network, also used by Temporal Transition Hierarchies[2] as mentioned briefly above, is the use of *higher-order* neurons. A higher-order neuron has incoming connections that are multiplicative instead of simply additive. That is, the input to a traditional hidden or output unit is:

$$\mathbf{in}^i = B^i + \sum_j w_{ij} x^j,$$

where $\mathbf{in}^i$ is the part in parentheses in Equations 3.1 and 3.2, and x^j is the value of input or hidden neuron j. On the other hand, the input to a *second*-order unit is the sum of second-order products (i.e., each term is the product of a weight and the value of two units):

$$\mathbf{in}^i = B^i + \sum_j \sum_k w_{ijk} x^j x^k. \tag{3.3}$$

The general case is the *sigma-pi* unit [88], where the products may be of any order:

$$\mathbf{in}^i = \sum_j w_{ij} \prod_{k \in S_j} x^k.$$

[2]Transition hierarchies also have a temporal component not present in the feed-forward networks considered in this chapter. Neural networks that can solve temporal tasks are presented in Chapter 4.

For each weight, w_{ij}, there is an associated set of units, S_j, by which the weight is multiplied. The maximum number of weights into unit i is equal to the size of the power set of units that can feed into i, since this represents one weight for every possible product.

If the input patterns are binary (even if the output is continuous), sigma-pi units can learn to compute any arbitrary mapping [29, 30]. It is appealing to use a single unit instead of an entire network of units, but a fully powerful sigma-pi unit would have 2^n terms if there are n units in the input vector. Two notable solutions have emerged to the exponential number-of-terms problem. The first solution is the reduction of order in the sigma-pi units. By limiting them to second-order terms only, functions impossible to compute with traditional units can be calculated while requiring only n^2 connection weights. This approach has been followed by Giles and Maxwell [35] and Pollack [74], described next. The second solution is the careful selection of terms that generate a neuron's input, described in Section 3.3.2.

3.3.1 Second-Order Networks

Giles and Maxwell's network was composed of zeroth-, first-, and second-order units expressed as follows:

$$O^i = f(w_i + \sum_j w_{ij} I^j + \sum_j \sum_k w_{ijk} I^j I^k). \tag{3.4}$$

The w_i weight is the same as a bias unit since it is not a coefficient of an input variable; it is the zeroth-order term. The w_{ij} weights are the traditional first-order weights, and the w_{ijk} weights are the second-order weights as in Equation 3.3.[3] Giles and Maxwell discussed a variety of learning rules for this network and observed a number of its interesting characteristics, for example, its generalization abilities and its invariance under certain groups of transformations.

Pollack's system uses back-propagation to calculate weight changes for a second-order network. His network splits the input into two parts, the standard input units, $I^1, I^2, \ldots, I^m$ and the *context* units, $C^1, C^2, \ldots, C^n$. The context units are used to calculate the weights for a second network, i.e., the weights of the second network are computed as linear sums of the C units:

$$w_{ij} = B_{ij} + \sum_k w_{ijk} C^k, \tag{3.5}$$

where B_{ij} is a bias unit.

These computed weights, w_{ij}, are then used as the weights of the second network (as described by Equations 3.1 and 3.2). The input to this second-order network is the vector of I units. Therefore, inputs to the hidden units of the second network look like this:

$$\begin{aligned} H^i &= f(B_i + \sum_j w_{ij} I^j) && (3.6)\\ &= f(B_i + \sum_j (B_{ij} + \sum_k w_{ijk} C^k) I^j) && (3.7)\\ &= f(B_i + \sum_j B_{ij} I^j + \sum_j \sum_k w_{ijk} C^k I^j), && (3.8) \end{aligned}$$

[3]The part of Equation 3.4 in parentheses will result from Equation 3.3 by removing the bias unit and instead adding an extra input unit whose value is always 1.

and these are, of course, the same second-order sigma-pi units of Equation 3.4, except that not all factors of input units are represented (i.e., there are no C^jC^k or I^jI^k products). Pollack observed enormous speedups over standard first-order neural networks in terms of the number of epochs seen. The cost of using a second-order network, however, is that the space and time complexities grow with n^3 instead of the n^2 of standard networks (where n is the number of units).

3.3.2 Partially Connected Higher-Order Networks

The second suggested solution to the number-of-terms problem with sigma-pi units is to carefully select the terms that generate a neuron's input. (This method is closest to the Temporal Transition Hierarchies method, which dynamically adds higher-order units one at a time.) This solution was suggested by Fahner and Eckmiller [30], who discussed two approaches toward this end (both called "parsiHON", for "parsimonious Higher-Order Neuron"). The first approach is to build a sigma-pi unit with *all* possible terms, and then to eliminate unnecessary ones. Though successful on small problems, this method still requires exponential space and time for the initial setup.

Fahner and Eckmiller's second method performs a stochastic search to find good sets of terms to include in the input equation. With this method, the architecture size (number of weights) is constant — fixed before training begins. A random set of terms is chosen and gradient descent is done to train the weights. (Training can be done quickly, since there are no hidden units.) After training, terms are removed according to the size of their weights: the larger the weight, the less likely it will be removed. This method is appropriate for much larger problem spaces than the first method and is similar to genetic algorithms [39] (which were incidentally suggested by Giles and Maxwell as a method of finding appropriate sigma-pi terms [35]). It is unclear how this method will scale, since the fraction of all possible terms that can actually be tested in a reasonable period of time shrinks exponentially with the size of the input space.

Both parsiHON methods resulted in excellent generalization. GAL, for example, which solved the two-spiral problem so quickly (above), generalized very poorly because it simply memorized the training set. ParsiHON, in contrast, captures much higher-order information and learned the spiral *pattern* (which extends well beyond the area covered by the training set).

Another technique that choses the number of terms during training and also seems to demonstrate quite good generalization is the tree-structured method introduced by Sanger [89]. This constructive approach creates new higher-order units during training which are combined to form basis functions over the input space. Though developed independently, Sanger's criteria for the construction of new units is much like that of Wynn-Jones (cf. Section 3.2) and of Temporal Transition Hierarchies (Section 6.2.4). In fact, of all methods described here, Sanger's is the most similar to Temporal Transition Hierarchies, though it is a feed-forward network and has no temporal component.

3.4 Conclusions

The approaches discussed above are effective measures for overcoming problems faced by simple feed-forward neural networks. Constructive approaches are useful when the optimal

network size is not known before training begins. Higher-order networks, on the other hand, learn in fewer training passes than standard first-order networks, and they can demonstrate improved generalization — but the cost is generally worse scaling behavior.

Table 3.1 summarizes the algorithms discussed in this chapter in terms of the dimensions of difficulty listed in Table 2.1 together with four characteristics especially descriptive of neural-network learning algorithms.

Training Algorithm	Dimension of Difficulty							Incremental	Constructive	Order > 1	Order > 2
	1	2	3	4	7	8	9				
BP	D	C	√	M/1	M−1	0	F	√	x	x	x
Upstart	L	B	√	M/1	M−1	0	F	x	√	x	x
GAL	D	B	√	M/1	M−1	0	F	√	√	x	x
Cascade Correlation	D	C	√	M/1	M−1	0	F	x	√	x	x
2nd order Nets	D	C	√	M/1	M−1	0	F	√	x	√	x
parsiHON	L	B	+	M/1	M−1	0	F	x	√	√	√
Sanger's Network	D	C	√	M/1	M−1	0	F	√	√	√	√

Table 3.1: Properties of the neural networks described in this chapter. The first group of columns corresponds to dimensions listed in Table 2.1, translating "sense" to "input", and "action" to "output". Since that table described environments and not algorithms, not all dimensions are appropriate. Dimensions 5, 6, 10, and 11 are meaningless for feed-forward neural networks. For dimension (1), "L" indicates that the algorithm requires locally encoded inputs or outputs, and "D" means that both inputs and outputs may have distributed representations. (2) "B" indicates that either the inputs or outputs must be binary, whereas "C" means that both inputs and outputs can have continuous values. (3) A check mark "$\surd$" indicates that the algorithm can learn any mapping from inputs to outputs (possibly by memorization). A plus sign "+" indicates that the algorithm can learn complex *functions* from inputs to outputs with particularly good generalization. (4) Since feed-forward networks have no state, all algorithms are "M/1" (many-one), meaning they can map many inputs to a single output pattern, but have no way to otherwise disambiguate the input information. More complex mappings require temporally sensitive networks. (7) "M-1" shows that the most complex underlying model that these algorithms can learn is Markov-1, since none of the algorithms retain information from previous inputs. (8) No history information is kept. (9) "F" indicates a fixed history (fixed at zero in these cases). The second group of columns is a set of properties descriptive of neural networks, indicating whether or not they can learn incrementally, build new units during training, have second-order weights, or have higher-order weights other than second-order. A check mark "$\surd$" indicates that the property describes the algorithm on that row, and "x" means that it does not.

4

Solving Temporal Problems with Neural Networks

Standard feed-forward neural networks are insufficient for solving many problems, specifically those with a temporal component. A feed-forward neural network implements a mapping from input units to output units, as was described in Chapter 3. In many robotics tasks, however, the robot's sensory input alone is insufficient to determine the correct action. There may be locations in the environment where the sensory information is ambiguous (a condition termed "perceptual aliasing" by Whitehead and Ballard [117], also known as the "hidden state" problem). If the environment is more complex than Markov-1, then its state, E(t) in Equation 2.5, is "hidden," since it cannot necessarily be deduced from the current sensory information alone. (Dimensions 4–10 of Table 2.1 are related to the environment's state). Those environments with a non-zero, finite number of hidden states are sometimes called *Partially Observable Markov Decision Processes* (POMDP's). A sufficiently demanding task in such environments requires the current state to be disambiguated using previous sensory information.

Whitehead and Ballard demonstrate how a robot can modify its own perceptual input by moving its sensors to complement its previous perceptual input with new information in order to determine its actual state. In many cases Whitehead and Ballard's technique is sufficient; however, if the robot is unable to alter its perceptions, or if by altering its perceptions it is still unable to disambiguate its sensory information, it may need to rely upon sensory information experienced earlier, much as we may navigate in a familiar house when the lights are out, or in an unfamiliar environment by remembering recent landmarks.

Another approach is that of Kuipers and Byun [51], whose agent builds an explicit topological map of its environment to distinguish among ambiguous perceptions. When the robot is in a position, say position A, and its perception there is identical to a perception recorded in, say, position B, the agent performs a *rehearsal procedure* that attempts to discover whether A is in fact the same position in the environment as B. This procedure visits neighboring locations in the topological map to test whether they match the neighbors of position B. If so, they are assumed to be the same. If not, A is incorporated as a new position in the map. As long as there is at least a *single* position in the environment where perceptual data is unambiguous, the agent can always theoretically determine when two positions are identical. This approach is promising but requires prior knowledge of the underlying sensorimotor apparatus, though Pierce [73] is making progress in removing this requirement.

There are also other cases when the robot might require more information than its current sensory inputs. Even if there are no two locations in the environment with identical sensory values, the robot's task may generate ambiguities. For example, the robot might be given one of several commands, each of which sends it through some common territory but toward a different goal location. While crossing the intermediate territory, the robot must remember the command it was given at the outset.

As will be shown in Chapter 6, Temporal Transition Hierarchies solve the hidden-state problem in Markov-k environments (where k is initially unknown) by building up behaviors that may last any arbitrary duration. The methods discussed in this chapter are other neural-network-based solutions to the problem of ambiguous sensory data. They are discussed here to provide background for later chapters and for purposes of later comparison with Temporal Transition Hierarchies.

4.1 Delay Lines

Hidden-state issues have been addressed with a variety techniques in the neural-network literature. The simplest of these is the use of *delay lines*. With a delay-line neural-network architecture, a window is kept of the past several sensory inputs from each sensor. If, as in Chapter 2, the vector of sensory inputs of length n at time t is $\vec{s}(t)$, then the jth component of $\vec{s}(t)$ is $s^j(t)$. The input to a delay-line neural network is:

$$s^0(t-\tau), s^1(t-\tau), \ldots, s^n(t-\tau), s^0(t-\tau+1), s^1(t-\tau+1), \ldots, s^n(t),$$

where τ is constant. This can also be expressed as

$$\vec{s}(t-\tau) \circ \vec{s}(t-\tau+1) \circ \ldots \circ \vec{s}(t),$$

where "$\circ$" is the concatenation operator. This architecture was used, for example, by NetTalk [98], which learned to map text to phonemes. The "sense" vector was a single binary-coded alphabetic character. In this system, τ was equal to seven, so seven characters were given to the system as input at every time step. The network learned to map the middle character $\vec{s}(t-4)$ to its phonemic category using the surrounding six characters as context.

A more sophisticated method for dealing with delay lines is the TDNN (Time-Delay Neural Network) described by Waibel [110]. This network is a hierarchy of sampled input over many time steps. The hierarchy is constructed with multiple layers of hidden units. Each hidden layer receives input from the layers below at the current time step and at the previous τ_l time steps, where τ_l can be different for each layer l. Thus, the input to layer l can be described as:

$$\mathbf{W}_l(\vec{x}^{l-1}(t-\tau_l) \circ \vec{x}^{l-1}(t-\tau_l+1) \circ \ldots \circ \vec{x}^{l-1}(t-1) \circ \vec{x}^{l-1}(t))$$

where $\mathbf{W}_l$ is the weight matrix connecting layer $l-1$ with layer l, and $\vec{x}^{l-1}(t)$ is the vector of values produced by input or hidden layer $l-1$ at time t.

The benefit of this kind of architecture is that information spread over many time steps can be integrated at each level of the hierarchy, allowing the highest levels of the hierarchy to compute functions over a large span of data. Nevertheless, the output is always a function of a fixed number of preceding inputs. If there is vital data not visible at the highest level, it cannot be incorporated into the output.

In robotics environments, delay lines would be useful if the current state could be disambiguated by knowing some fixed, finite number of previous states. For example, in Markov-k environments (see section 2.2.3), the current state of the robot can be identified unambiguously by examining the last k input vectors. Figure 4.1 is a Markov-2 environment. In this

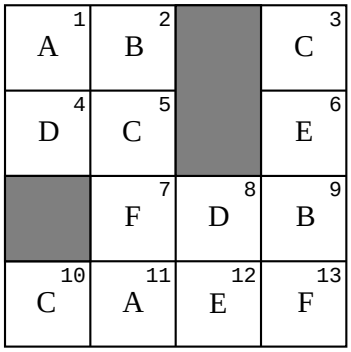

Figure 4.1: A Markov environment that requires the current and previous input to determine the current state.

environment, there are thirteen different positions in which the robot can land, but there are only six distinct sense vectors (labeled A–F). Yet if the robot moves from any one cell to any adjacent cell, its position will be uniquely determined. If, for example, the robot sees D at one time step and E at the next, it must be in cell twelve. If it were to see A instead of E in the second time step, it would be in position one.

If the robot lives in a Markov-k environment (and its task is sufficiently demanding), then it must have the information from its last k sense vectors to choose the correct action in all situations. Having less information than this at its disposal would cause ambiguities and force errors.

4.2 Learning Time Delays

Because there are Markov-k environments where k is initially unknown, some algorithms have been developed that learn the amount of history information needed. As mentioned above, one such algorithm is Temporal Transition Hierarchies, which will be described in Section 6.2. Another is Bodenhausen and Waibel's system, Tempo 2 [13]. In Tempo 2, input units have time delays that can be learned as adjustable parameters. Each input unit, in fact, has three adjustable parameters for every incoming connection: the weight, time delay, and width of the time delay's receptive field. The receptive field is a Gaussian-shaped "window" in time that responds to input lines at a particular time delay in the past.

When gradient descent is done and these parameters are modified for each connection, the network learns to respond to events that occurred in the past and learns for itself *how far into the past* it needs to look to get this information. Therefore, if a Markov-k task is to be learned, this network can theoretically learn to span k steps to compute the proper output. Furthermore, though initially each unit responds to each input line through a single window, new windows can be created automatically when needed so that the units can respond to every input line at any number of time delays.

A related architecture, designed by Day and Davenport [19], responds to discrete intervals of time in the past rather than to intervals convolved with a Gaussian. Their architecture allows a prespecified number of adaptable time delays to appear for *any* connection in the network (not just connections to the input units). To develop their network, they also

explicitly formalized the notion of time delays and produced a method for doing gradient descent with respect to the parameters that specify the delays.

4.3 Recurrent Networks

Like Temporal Transition Hierarchies and the algorithms just mentioned, recurrent neural networks can also be used in Markov-k environments where k is not specified in advance. The topology of a non-recurrent network is acyclic. In those "feed-forward" networks, each unit sends information to other units from which it will never directly or indirectly receive information. In recurrent networks, on the other hand, there are cycles. Some units send information to other units that they either directly or indirectly receive information from themselves. All cases to be considered here are *discrete-time* networks in which information is sent over each connection exactly once per time step (i.e., each unit is updated once per time step). The simplest recurrent networks look just like the feed-forward networks described in Chapter 3, except that some hidden units have single, recurrent self-connections, and there are no other recurrent connections. Examples of these networks will be described next: First, Mozer's *Focused* Back-Propagation network [65] and Bachrach's *Sticky-bit* network [4]; and Second, Fahlman's RCC (Recurrent Cascade Correlation) architecture [27]. After these comes a description of fully connected recurrent networks.

4.3.1 The *Focused* and *Sticky-bit* Architectures

The focused architecture uses hidden units with adjustable connections in the hidden layer to encode a variable decay rate. The output of each unit is described as the squashed sum of its input plus an adjustable fraction, d of its previous value:

$$H^i(t) = f^i(\sum_j I^j(t)w_{ij}) + d^i H^i(t-1), \tag{4.1}$$

where $H^i(t)$ is the value of the ith hidden unit at time t, and $I^j(t)$ is the value of the jth input unit at time t. A unit keeps a kind of running average of its past inputs, which in principle allows the network to make use of events that occurred in the arbitrarily distant past. The focused network operates by keying into certain features of the input and then remembering these features until later.

The units in Bachrach's sticky-bits network include the $d^i H^i$ term within the squashing function.

$$H^i(t) = f^i(d^i H^i(t-1) + \sum_j I^j(t)w_{ij}). \tag{4.2}$$

Bachrach also uses a sigmoidal activation function with asymptotes at -1 and $+1$, allowing the H^i's to "stick" at a positive or negative value when the input to the unit is strongly positive or strongly negative respectively and d^i is large. With this architecture, large input values can cause a hidden unit to "remember" that an event has occurred in the past and to keep that value available as processing continues until it is needed at the output. Since it is stable in two possible states, each hidden unit can detect the presence or absence of a single feature [4].

Learning in both networks is done by gradient descent in the error space with respect to the modifiable parameters w_{ij} and d^i for all i and j. The networks are more powerful

than the adaptive delay-line networks (as well as Temporal Transition Hierarchies) in that they can in principle remember an event over an arbitrarily long time span. The amount of information that the nets can store is only limited to the number of hidden units.

Unlike delay-line models, the Mozer and Bachrach networks do not store a history of previous values one after the other to be used at later time steps. For example, say a network is to be trained to simulate a queue, specifically to produce as output a copy of the network's input twenty steps earlier. At every step the input changes, and twenty time steps later these changes must be reflected in the output (i.e., $\text{Output}(t) = \text{Input}(t - 20)$). Obviously, a delay-line model is ideal for this task, since it is the nature of delay lines to reproduce a signal after a certain duration. While the focused network architecture is theoretically capable of solving this task for arbitrary delays given discrete valued inputs, weights of arbitrary precision, and a powerful enough transfer function (see Appendix A for a differentiable solution), such a network is highly impractical and not possible with limited precision hardware or with monotonic transfer functions.

4.3.2 Recurrent Cascade Correlation

The RCC (Recurrent Cascade Correlation) network resembles the sticky-bit architecture but is a constructive network that adds units as needed just as the Cascade Correlation architecture does in non-temporal domains (Section 3.2). The output of the RCC hidden units is the same as in Equation 4.2, with the decayed value inside the transfer function. That is, the network's hidden unit's receive input from units lower in the network at the current time step and from themselves at the previous time step. Again, gradient descent is performed to tune the weights and the decay parameters. However, in RCC, a pool of new units are trained *en masse* to predict the output error of the network. The best one is taken and integrated into the network; its weights are frozen; and its output is made available to the next group of trainees to assist them in correcting the remaining error. The network thus grows one hidden layer at a time with a single unit in each layer. The output values from all units feed into every unit higher in the network.

RCC can learn to extract any arbitrary number of features from the input string, keeping them for any arbitrary duration. These networks should be able to learn any Markov-k task where k is initially unknown, since they are capable of storing an arbitrary amount of past history information. However, there are computations these networks cannot perform. This has been proven by Chen, *et al.* [15], who showed that RCC networks using sigmoid or threshold activation functions are unable to learn certain classes of finite-state grammars. (This is due to the fact that none of the RCC hidden units have connections downwards, back to lower units — units closer to the inputs.) Another limitation of RCC is that it is a "batch" learning algorithm. Training must be done on a fixed set of training patterns and cannot be done incrementally as new data is presented. (Comparisons with RCC are shown in Section 7.2.1.)

4.3.3 Fully Connected Recurrent Architectures

In contrast to focused and RCC networks, certain classes of fully connected recurrent networks (networks that allow cycles in their connectivity besides just self-connections) have been shown by Minsky [61] to be Turing equivalent. Limited versions of these networks have

been devised by Jordan [43] and Elman [25]. These networks, though capable of computing functions not computable by focused and RCC networks, nevertheless have limitations in what they can *learn*, due to the fact that their learning algorithms only approximate gradient descent. Other networks do compute the complete gradient and are capable of learning as well as solving difficult problems. Back-Propagation Through Time and its variants [88, 115, 121] treat the temporal characteristics of sequential tasks *spatially*. These algorithms create a very large virtual network by replicating the real network once for every time step and then attaching these networks together. The outputs of the network at one time step are fed as inputs to the network at the next time step. At some point — either at the end of the sequence or after a certain number of time steps — forward propagation through the chain of networks is stopped, and back-propagation through the entire *virtual* network is performed. The weight changes can be applied to the weights immediately or after an epoch of such sequences.

Back-Propagation Through Time performs exact gradient descent when the virtual network spans the entire sequence. It is not an incremental learning algorithm, and infinite sequences are therefore impossible to learn. One way to train on infinite sequences is to back-propagate over only the past n time steps — where n is a carefully chosen integer for the task to be learned — and to ignore all previous time steps. Williams and Peng [121] discuss the positive and negative consequences of this approach.

A different, incremental method [85, 114, 123] calculates the complete gradient as a function of the derivatives computed at the previous time step. This approach, termed RTRL (Real Time Recurrent Learning) by Williams and Zipser [123], does not require the large virtual network that grows according to the length of the sequence to be learned. No values from previous time steps (other than the trace of derivatives) must be stored for the computation. (Comparisons with this method are shown in sections 7.2.1 and 7.2.2.) But RTRL requires $O(n^3)$ storage space (where n is the number of neurons), and $O(n^4)$ computations at every time step. Other algorithms [95, 102, 121] have been proposed that attempt to reduce the number of computations per time step to $O(n^3)$. Though the cost of using a fully recurrent network remains high, numerous studies have demonstrated very intelligent recurrent-network behavior learned via gradient descent [122, 123]. These networks have been shown capable of solving very difficult temporal tasks, though generally extremely slowly. They have also been shown to be particularly poor at learning long temporal contingencies, i.e., long time delays. As will be seen in Section 7.2.2, Temporal Transition Hierarchies can learn long time delays very quickly.

Multiscale Temporal Networks. To address the problem of learning long temporal contingencies, Mozer [66] created a network that integrated aspects of his focused network into a fully connected network. These networks added a special connection from each hidden unit to itself with a built in decay rate. The decay rates were fixed before training. Mozer found that if the decay rates were set properly, very long delays could be learned. (Comparisons with this method are shown in Section 7.2.2.) However, as the time delays increase, so does the network's sensitivity to the decay rate.

4.3.4 Second-Order Recurrent Networks

Because of the weaknesses of back-propagation through time, its equivalents, and its variants, some more powerful recurrent networks have been proposed. In particular, higher-order recurrent networks can solve very difficult tasks while converging after fewer training examples than some classes of standard recurrent networks [36, 60, 74]. These networks not only have additive connections between units, but also have multiplicative connections. Though these networks do not have a great deal in common with the methods proposed in this dissertation, they are discussed here for completeness.

The network described by Pollack [76] is given below. There are three differences between this network and its non-recurrent counterpart [74] described in Section 3.3. First, each input and output value is time indexed (e.g., $O^i(t)$ is the value of output unit O^i at time t). Second, the *context* units of the previous version are now the output units from the preceding time step. Third, there are no hidden units. Otherwise, there is a direct correspondence between Equations 3.5–3.8 and Equations 4.3–4.6.

$$w_{ij}(t) = \sum_k w_{ijk} O^k(t-1) \tag{4.3}$$

$$O^i(t) = f(\sum_j w_{ij}(t) I^j(t)) \tag{4.4}$$

$$= f(\sum_j (\sum_k w_{ijk} O^k(t-1)) I^j(t)) \tag{4.5}$$

$$= f(\sum_{j,k} w_{ijk} O^k(t-1) I^j(t)) \tag{4.6}$$

(The fact that the non-recurrent network had explicit bias units and the above description does not is really not a difference, since bias units can be added to the framework above by simply clamping an input unit and a context unit to 1.0.)

Pollack was interested in the task of learning to recognize finite-state grammars. Giles, *et al.* [33], and Watrous and Kuhn [113] both used Pollack's architecture, augmenting it by doing full gradient descent in the network's three-dimensional weight matrix with respect to the error generated over multiple time steps. In fact, the network used by Giles, *et al.*, is incremental in that it computes the derivatives of the weights as a function of the derivatives computed during the previous time step — just as is done by RTRL for first-order networks. These networks learned to recognize complicated grammars from small numbers of sometimes ambiguous training sets. The training sets contained both positive and negative examples and were ambiguous in that they sometimes suggested more than one correct finite-state grammar. The importance of this work is in the difficulty of the tasks that were solved. Table 4.1 lists the grammars learned.

Some of these are extremely difficult. Even the first, the simplest, cannot be learned by a delay-line network, since the network must record whether it has seen a zero and remember this for an arbitrary period of time.

Bachrach [5] designed a related architecture strictly for the purposes of learning finite-state automata in a connectionist system. His architecture was modeled after the finite-state-machine learning algorithm of Rivest and Schapire [84]. Bachrach's network is capable of learning some extremely difficult grammars from positive examples only. Bachrach's network, named SLUG, has a different weight matrix for every input unit. (Only one

1*
(1 0)*
no odd *zero* strings after odd *one* strings
no 000's
pairwise, an even sum of 01's and 10's
number of 11's - number of 0's = 0 mod 3
0*1*0*1*

Table 4.1: Regular languages from Tomita [109] used by Pollack [76], Giles, *et al.* [33], and Watrous and Kuhn [113] for teaching higher-order recurrent networks to recognize finite-state grammars. Both positive and negative examples were used for training.

input unit can be active at a time.) His network allows recurrent connections, so the hidden units can propagate their values to each other from one time step to the next through the weight matrix selected by the input node. Gradient descent is done with a variant of Back-Propagation Through Time: as in standard Back-Propagation Through Time, a large virtual network is constructed (Section 4.3.3), but unlike the standard case, the weight matrices of these networks can be different at different time steps, and weight changes must be applied to the weight matrix for which they were computed. SLUG, it turns out, is actually identical to Pollack's network (and the network that Giles, *et al.*, and Watrous and Kuhn have used) except with the explicit restriction that the inputs are locally encoded (only one is on at a time), and the function f is the identity map. (This equivalence is shown in Appendix B). Therefore, the results that Bachrach showed for SLUG — including its relationship to the work of Rivest and Schapire [84] — apply to recurrent second-order networks in general. In particular, Bachrach compared his system on difficult tasks to many traditional networks, including recurrent networks that used Back-Propagation Through Time, and he found a great improvement. The standard networks were incapable of learning even his simplest tasks. Bachrach suggests in his thesis [5, p. 65], that one of the key advantages of his network is the multiplicative property of the connections. The conclusion to be drawn from this is that, even though many classes of recurrent networks can theoretically perform any computable temporal task [23, 61], higher-order networks seem to be far superior *learners* of difficult finite-state grammars. Temporal Transition Hierarchies also have higher-order connections (though they're not recurrent). The fast learning times that will be demonstrated in Chapter 7 are in a large part due to these higher-order connections.

The recurrent second-order networks are very powerful, but they scale very poorly. Those which implement incremental learning have a space complexity of $O(n^4)$ and a time complexity of $O(n^5)$ at each step, where n is the number of units.

Constructive Fully Recurrent Networks. After describing the limitations of the RCC network, Chen, *et al.* [15], went on to design a constructive network of their own. Theirs adds units as learning progresses to a fully connected, second-order recurrent network (that of Giles, *et al.* [34, 33], discussed above). Since it is fully connected, it does not suffer from the problems faced by RCC and is theoretically Turing equivalent, though it still suffers from the poor scaling behavior of second-order recurrent networks.

Training Algorithm	Dimension of Difficulty							Incremental	Constructive	Order > 1	Order > 2
	1	2	3	4	7	8	9				
TDNN	D	C	√	M/M	M−k	k	F	√	x	x	x
Tempo 2	D	C	√	M/M	M−k	k+	F	√	√	x	x
Day/Davenport	D	C	√	M/M	M−k	k+	F	√	x	x	x
Focused	D	C	√	M/M	⊂FSA	k	∞	√	x	x	x
RCC	D	C	√	M/M	⊂FSA	k+	∞	x	√	x	x
BPTT	D	C	√	M/M	FSA	k	∞	x	x	x	x
RTRL	D	C	√	M/M	FSA	k	∞	√	x	x	x
Multiscale	D	C	√	M/M	FSA	k	∞	√	x	x	x
2nd order Rec.NNs	D	C	+	M/M	FSA	k	∞	√	x	√	x
Contructive Rec.NNs	D	C	+	M/M	FSA	k+	∞	√	√	√	x

Table 4.2: The characteristics of the temporally sensitive networks discussed in this chapter. The columns are the same as those of Table 3.1, but, since these algorithms can keep state information, this table has some differences. Dimension (3) indicates the current-sense→action mapping given that the previous state is known. (4) Since all of these networks are capable of using context to map ambiguous sensory inputs to unambiguous state representations, all are marked "M/M" (many–many). (7) "M-k" means the algorithm can learn Markov-k sequences; "FSA" means they can represent any FSA; and "⊂FSA" means they are more powerful than Markov-k but cannot represent arbitrary FSA. (8) "k+" means the algorithm can learn for itself the amount of data it needs. (9) "∞" denotes that state information can be held indefinitely. All other entries are as in Table 3.1.

4.4 Conclusions

The rows of Table 4.2 list the neural-net architectures described in this chapter. The columns list the important features of those architectures. In general, recurrent networks and other approaches to learning temporal tasks have great limitations. Specifically, they are either incapable of solving any but the simplest tasks, or they are dreadfully slow. Second-order recurrent networks are capable of learning complex grammars, but by sacrificing either speed or incremental learning. Because of their poor scaling behavior, it is unreasonable to try using them in problems with more than just a few input and output units.

Temporal Transition Hierarchies, the solution I propose in Chapter 6 is a non-recurrent, higher-order network that learns temporal tasks quickly and incrementally while constructing new units hierarchically, but the price is its limitation to learning Markov-k environments where k is originally unknown.

5

Reinforcement Learning

I discussed reinforcement learning briefly in Section 2.1.3. This chapter contains a description of the most common reinforcement techniques, which attempt to resolve the issues surrounding dimensions 10 and 11 in Table 2.1. It begins with reviews of the AHC (adaptive heuristic critic) [103, 106], and Q-learning [111], though they are both currently very popular and their details are known to many. Q-learning will be used in Section 7.3 to provide the framework for CHILD. The chapter then describes some research relating reinforcement learning to dynamic programming, followed by a discussion of reinforcement-learning research using gradient-descent methods. If you are already familiar with any of these techniques, you will probably want to skip the corresponding sections. Finally, I will try to explain the field from a slightly more intuitive perspective and then explore a few resulting observations.

5.1 The Adaptive Heuristic Critic

Reinforcement-learning tasks assume the existence of an agent. The agent receives sensory data as input and generates actions as output, just as with the robots described in Chapter 2. Occasionally the agent takes an action that results in a reinforcement. The reinforcement is either rewarding or punishing. The AHC is a general architecture that increases the probability of actions that lead to greatest reward and decreases the probability of actions that lead to punishment. More formally, the AHC attempts to maximize the agent's long-term reinforcement [9, 103, 105]. This architecture divides the agent into two modules: one that chooses an action and one that estimates the agent's future reinforcement given its current input. Thus, in terms of Table 2.1, one part addresses dimension 5, mapping states to actions, and the other addresses dimension 10, mapping state/action pairs to reinforcement predictions. The former is called the policy module. The latter is called the critic module. A picture is given in Figure 5.1.

In a typical AHC task, the agent's environment is Markov–1; i.e., the agent's state in the environment is encoded unambiguously in the agent's sensory input (see Section 2.2.3). The input and actions are often encoded locally in a binary vector (exactly one item will have a value of 1.0, and all others will have a value of 0.0), but this restriction is not necessary; it merely serves to simplify the learning task and highlight its reinforcement-learning aspects. Learning without this restriction has also been done [3, 54, 92, 116]. The only *real* restriction with the AHC (and with most other reinforcement-learning techniques as well) is that the learning algorithm used by the policy and critic modules must be capable of distinguishing the underlying environmental states. That is, besides learning the state→action and state/action→reinforcement-prediction mappings, it must also be capable of learning the sense→state mappings.

For simplicity, the following description assumes the state of the agent at the current time step, s_t, is given to the agent as input (i.e., s_t is the sensory vector, $\vec{s}(t)$ in Section 2.1).

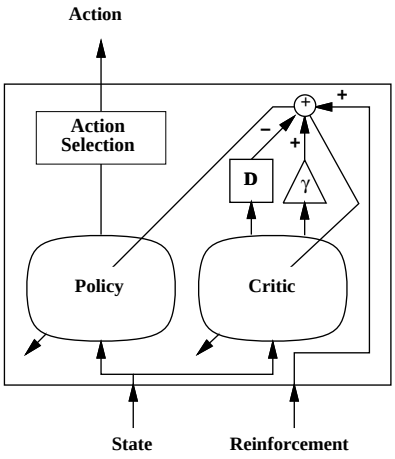

Figure 5.1: The Adaptive Heuristic Critic [106] has two components, the policy module, which generates action suggestions from the current senses, and the critic module which predicts discounted future reinforcements. The policy module produces a value for each action that determines the probability the action will be chosen. (An action is then chosen according to these probabilities by an action-selection mechanism.) The current reinforcement plus the discounted critic's value (multiplied by the discount factor, γ) is compared with the critic's value at the previous time step (the box labeled "D" delays its input by one time step); the difference is used to correct the critic and to modify the policy module. Implementations for this are given in Section 5.1.1.

The AHC works as follows: an agent exists in a state and in that state takes an action. The mapping from states to their respective actions is called the agent's *policy*. If the agent always takes the same action every time it visits a state, the policy is *deterministic*. If in some states the agent instead chooses different actions with different probabilities, the policy is *stochastic*. The policy is therefore the function f in Equation 2.8. This function is implemented by the policy module.

The critic module, on the other hand, predicts the *discounted future reward* (*dfr*) for each state given the current policy. The *dfr* is the expected sum of all future rewards, each discounted by a certain amount (the discount factor) depending on how far into the future the reward will be received following the current policy. That is,

$$V(x) = E[\sum_{k=0}^{\infty} \gamma^k r(k)],$$

where $V(x)$ is the discounted future reward of state x; $E[.]$ denotes the expected value (which is necessary for stochastic environments, where the next state is not a deterministic function of the current state and policy); γ is the discount factor; and $r(k)$ is the reward the agent will receive for taking the k^{th} policy action after visiting state x.

For example, if the environment and policy are both deterministic, and if by following a certain policy the agent will receive a reinforcement of 1.0 for taking policy action a in state z and will receive no other reinforcement, then $V(z)$ will be 1.0. If $\gamma = 0.9$, then $V(y)$, where state y leads in one step to state z, will be 0.9; and $V(x)$, where state x leads in one

step to state y, will be 0.81, and so on. If all reinforcements are finite and $0 \leq \gamma \leq 1.0$, then even if the agent continues forever, the discounted reward for every state is finite [9].

A distinction can be made between two types of tasks: those that terminate, and those that do not. In terminating tasks, the agent is stopped when it reaches a *halting* state, or when it executes the task for some predetermined maximum number of time steps. Once the agent is stopped, it may be started again on the same task. A new *trial* is said to start each time the task is begun again. In terminating tasks, the agent can only visit a finite number of states.

For non-terminating tasks, there is only one trial. The agent is never expected to complete the task and can therefore reach an infinite number of states (provided the number of reachable states is infinite). Though non-terminating tasks are closer to the real world, it is assumed in the following section that tasks are terminating (though they may proceed for arbitrarily long) and that multiple trials may be performed to train the agent.

The AHC can be trained with the following successive approximation procedure. A deterministic policy is used. On every trial, the policy is followed and the critic's value for each state visited is re-computed. After every action, the critic's estimate of the previous state's *dfr* is modified to reflect the reinforcement just received together with the critic's estimate of the current state's *dfr*:

$$Critic(s_t) \Leftarrow r(s_t, a_t) + \gamma Critic(s_{t+1}), \tag{5.1}$$

where $Critic(s_t)$ is the critic's current estimate of $V(s_t)$; $r(s_t, a_t)$ is the reinforcement received for taking action a_t in state s_t; and "A$\Leftarrow$B" means that A is modified so as to reduce $|B - A|$, the absolute difference between A and B.[1] This is a form of *temporal difference learning* [104]. Over successive trials, the estimates converge under appropriate conditions to the correct values [112, 22], i.e., the left-hand side of Equation 5.1 will converge to the expected value of the right-hand side.

Once a critic has been trained the (deterministic) policy can be improved by taking random non-policy actions and comparing the discounted future rewards of the states visited. If the agent takes a non-policy action, a, from state j to state i, and the reinforcement received plus the *dfr* of state i is better than expected (i.e., $r(i, a) + \gamma V(i) > V(j)$), then action a is better than the policy action. A new policy can then be created that incorporates the improved action. If the environment is stochastic, the new action must be tried enough times (in theory, an infinite number of times; in practice, less) to ensure that it actually improves the policy in the average case. A new critic module can now be trained to predict the correct *dfr* values for this new policy. Clearly, only states that lead eventually to state j will be affected. Once a new critic is in place, more improvements to the policy can be learned.

The policy can only *improve* during policy modification (i.e., the mean *dfr* over all states can only *increase*), and, after retraining, the critic module will reflect the actual *dfr* for the new policy. Therefore, this process of alternatively improving the policy and the critic never produces a new policy that had been the policy previously. That is, the process of modifying the policy always increases the mean *dfr* until eventual convergence [9, §5.2]. In the final

[1] How this adjustment is made depends on the mechanism for representing A (e.g., look-up table, neural network, etc.)

policy, no state→action mapping can be changed to increase the policy's mean *dfr*. This policy is globally optimal and is called an *optimal* policy.

Instead of adjusting the policy and critic iteratively, these processes can also occur simultaneously, by randomly taking non-policy moves. In this case, the policy is stochastic, so a *weight* is associated with every action in each state to determine its likelihood of being chosen. The weight increases or decreases depending on whether this move generates a higher or lower *dfr* than the other actions in that state. Therefore, if a move results in a *dfr* better than predicted, it becomes more likely to be chosen in that state in the future. If it results in a lower *dfr* than predicted, it becomes less likely to be chosen in the future. The next action is chosen stochastically by the action selector (Figure 5.1) based on the weights of the actions available from the current state. Unlike the iterative approach, the simultaneous case has not been proven to converge to the optimal policy.

5.1.1 Implementation

The critic and policy modules are easily implemented as look-up tables. Simultaneous learning can be done with the following learning rules:

$$\Delta_{a_t} = r(s_t, a_t) + \gamma Critic(s_{t+1}) - Critic(s_t) \tag{5.2}$$

$$w(s_t, a_t) \leftarrow w(s_t, a_t) + \alpha\Delta_{a_t} \tag{5.3}$$

$$Critic(s_t) \leftarrow Critic(s_t) + \beta\Delta_{a_t}, \tag{5.4}$$

where $w(s_t, a_t)$ is the "weight" given to action a_t in state s_t, α is a learning-rate parameter for the policy module, $r(s_t, a_t)$ is the immediate reinforcement the agent receives for choosing action a_t in this state, and β is a learning-rate parameter for the critic. These rules state what was expressed above: the critic is modified so as to predict the *dfr* better. The weight for action a_t in state s_t increases/decreases if the expected *dfr* is better/worse than expected. The probability of choosing an action is determined from the weights using the following Gibbs distribution:

$$P(a|s_t) = \frac{e^{w(s_t,a)}}{\sum_a e^{w(s_t,a)}}. \tag{5.5}$$

The probability of choosing action a in a state is proportional to the weight of a relative to the weights of all other actions, where the weights are magnified (exponentially) to accentuate their differences.

Alternatively, the critic and policy modules are often implemented as neural networks. Lin [54], for example, used one network for the critic and separate policy networks for each action. Each network takes the current sensory vector as input. The critic network is trained to predict the *dfr* for each input. The policy network for each action is trained to predict the weight values for that action. This is done by using Δ_{a_t} as the error value to back-propagate through network a_t with s_t as the network input. Neural networks are more useful than look-up tables when the sensory vectors are large and are not locally encoded, and where the mapping from inputs to actions can support meaningful generalization. Lin's tasks used a large, complicated sensory input which allowed many opportunities for generalization.

5.2 Q-learning

The AHC is capable of learning many difficult reinforcement tasks, but in its standard, simultaneous form, it has not been proven to converge to the optimal policy. *Q-learning* [111] is a reinforcement-learning method that *has* been proven to converge under certain conditions [112, 111]. It is used in Section 7.3 as the reinforcement learning component of CHILD.

Q-learning is actually very similar to the AHC. It, in a sense, combines the functions of the two AHC modules into a single module. The new module makes predictions of the *dfr* not for each state, but for each state/action pair. At every step the state/action pairs for the current state are examined; the action with the highest estimated *dfr* is the one most likely chosen. The *dfr* of the action actually chosen is then increased if the estimated *dfr* of the next state is better than predicted and decreased if it is worse. (The *dfr* of a state is the *maximum dfr* over all actions available from that state.)

The Q-learning update rule is formalized as follows:

$$Q(s_t, a_t) \stackrel{\leftarrow}{=} r(s_t, a_t) + \gamma(\max_a Q(s_{t+1}, a)), \tag{5.6}$$

where $Q(s_t, a_t)$ is the estimated *dfr* (the "Q-value") for taking action a_t in state s_t, the agent's state at time t. Actions may be chosen again according to the following Gibbs distribution:

$$P(a|s_t) = \frac{e^{\frac{Q(s_t,a)}{T}}}{\sum_a e^{\frac{Q(s_t,a)}{T}}}, \tag{5.7}$$

where T is a *temperature* parameter that decreases in an annealing process. As $T \to \infty$, the action selection becomes entirely random; as $T \to 0$, randomness plays no part and the choices become deterministic. When table look-up is used and T decreases asymptotically (to make sure there is always a non-zero chance of trying every action in every state), the Q-values are, theoretically at least, guaranteed to converge to their correct values. (In practice, this would require infinite training.)

Q-values can be maintained in a look-up table, or, as with the AHC, they can be stored in a neural network. Lin [54], for example, used a separate network for each action, updating them with the following learning rule:

$$\Delta Q_{a_t} = r(s_t, a_t) + \gamma(\max_a Q(s_{t+1}, a) - Q(s_t, a_t)), \tag{5.8}$$

where ΔQ_{a_t} is the error value to back-propagate through the network corresponding to the action just taken. Lin found that when implemented with neural networks, the AHC and Q-learning systems had comparable performance. Though Q-learning is guaranteed to converge when exact values are stored and retrieved (e.g., lookup tables), no such guarantee can be made when the values are approximated, as they are with neural networks.

5.3 Dynamic Programming

Both Q-learning and the AHC are grounded in the theory of dynamic programming. In fact, reinforcement-learning problems can in general be cast as dynamic-programming problems. The result is that the well understood methodologies of dynamic programming

can be used for reinforcement learning.[2] This includes all instances of reinforcement learning with discrete actions, where complete state information is given, and when the environment is a Markov Decision Process (Section 2.3). One subset of these is the case of finding the best path from a set of starting states to a set of goal states (similar to traditional AI heuristic search). These problems — where actions leading to a goal state result in a positive reinforcement and all other actions receive no reinforcement — are a subset of the dynamic-programming problems known as *shortest-path problems* [11, 12ff][12]. Dynamic programming can apply to much more sophisticated reinforcement schemes than shortest-path problems, however, including cases where the environment's transitions are stochastic and where each action may receive positive or negative reinforcement.

The formula that summarizes dynamic programming is the Bellman Optimality Equation, which can be stated as:

$$f^*(i) = \max_{a \in A(i)} \left[r(i,a) + \gamma \sum_{j \in S} p_{ij}(a) f^*(j) \right], \tag{5.9}$$

where $f^*(i)$ is the *optimal evaluation function* applied to state i; $r(i,a)$ is the reinforcement for taking action a in state i; $A(i)$ is the set of actions available in state i; γ is the discount factor; and $p_{ij}(a)$ is the probability of arriving in state j after taking action a in state i. In the deterministic case, the probability of arriving at a particular state given a chosen action is always 1 or 0. In the stochastic case, taking an action in a state does not necessarily determine the state that will be reached. The critic is analogous to the evaluation function, though the critic produces the *dfr* values for the *current* policy while the evaluation function produces the *dfr* values for the *optimal* policy.

The Bellman Optimality Equation defines an evaluation function whose value in each state depends upon its value in other states, thus constituting a set of simultaneous non-linear equations that can be solved iteratively through a successive approximation procedure. A notion central to the iterative solution of the equation is the idea of *backing up* a state. A backup is the re-estimation of a state's value as a function of the states reachable from it (i.e., computing new approximations to $f^*(i)$ based on the most recent estimates of $f^*(j)$ for all states j reachable from i).

In reinforcement-learning tasks, the problem is to decide which actions are the best, i.e., to determine the optimal policy. This can be done by solving the Bellman Optimality Equation to determine the evaluation of each state, and then to take the actions that maximize the expected evaluation of the next state. This is trivial if the evaluation function, the reinforcements, and the probability distribution are known. The optimal action in state i, a_i^* is simply the action that maximizes the right hand side of Equation 5.9, i.e.,

$$a_i^* = \operatorname*{argmax}_{a \in A} \left[r(i,a) + \gamma \sum_{j \in S} p_{ij}(a) f^*(j) \right],$$

where $\operatorname{argmax}_a(f(a))$ returns the argument, a, that maximizes $f(a)$. If more than one action maximizes the equation, any such action can be chosen. Though an optimal policy can be determined from the optimal evaluation function, the converse is not the case.

[2]For more information, see the excellent synthesis given by Barto, Bradtke, and Singh [11].

One way to discover the optimal policy is through the AHC method discussed above, which approximates a form of dynamic programming known as *policy iteration* [105] in that it successively modifies the policy until the optimal policy is reached. The optimal policy can also be determined through *value iteration* methods, such as Q-learning, which solve Equation 5.9 by successive approximation. The advantage of policy iteration is that it can discover an optimal policy long before correctly calculating the optimal evaluation function. In contrast, value iteration solves Equation 5.9 without recognizing when an optimal policy has been determined [11]. This should not be construed, however, as indicating that value iteration offers only one way to solve the equation. Many different methods are described in the reinforcement-learning literature, each with its own advantages and disadvantages. One such method is RTDP (Real-Time Dynamic Programming) [11], which backs up the agent's current state together with as many other states as time permits at each step. Special cases of this are the Prioritized Sweeping [64] and the Queue-Dyna [71] algorithms, which at every step modify the values of those states where the modifications are the largest. The agent can thus make the best use of its limited time between steps.

5.4 Gradient Following Methods

Dynamic programming methods can be extended with model learning to improve speed, but it is limited to cases where states and actions are discrete. Gradient-following techniques for determining the optimal action when the actions are continuous have been explored by Werbos [116], Munro [68], Jordan and Jacobs [45], Schmidhuber [91, 93], Thrun, *et al.* [108], Thrun and Möller [107], Bachrach [5], Linden and Weber [57], and others. These methods are all based on the concept of using a neural network to learn a model of the environment and then doing gradient descent in this differentiable model to improve the quality of the agent's decisions. There are two clear categories of work done in this way. The first is that of *network inversion*, the second is that of *controller modification*.

Network inversion is the method of using gradient descent to do constraint satisfaction not on the network weights, but on the network inputs. This technique has been used by Williams [119], Kindermann and Linden [49], Linden and Weber [57], Hwang and Chan [41], and others, many of whom have independently developed the same technique while trying to solve the inverse-kinematics problem. The central notion is that of taking the partial derivative of the error with respect to the network's inputs, then successively modifying these inputs to reduce the error until a minimum is reached. The result is a pattern which, when given to the network as input, will generate an output most closely matching the target. If, for example, a network can be trained to produce as output the position of a robot's hand given as input the joint angles of its arm, then the network can be inverted in hopes of discovering the joint angles that will put the robot's hand in a specified location.

There are two unfortunate drawbacks of applying the inversion process, however. The first is that of spurious local minima [49]. The network may converge to an input that generates an output not particularly close to the desired output (just closer than any of its neighbors in input space). The second problem, closely related to the first, is that there is often a many-to-one relationship between the input patterns and the target patterns. The inversion of the network averages among these global minima to produce an interpolation of those that generate this target [41, 44, 46, 56].

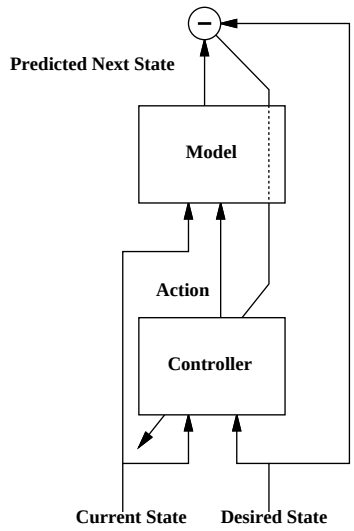

Figure 5.2: The controller produces an action given the current state and the desired state. The prediction of the model is then compared with the desired state, and the difference is back-propagated through the model and into the controller where the weights are changed.

One solution to these problems of multiple minima, proposed by Kindermann and Linden, is to "erase" spurious local minima by adding specially designed training patterns to the training set. Another solution is to impose extra constraints so that only a single valid input will be found [41], or so that only one input can minimize all the imposed constraints (a technique used by Jordan [44] for controller modification but not network inversion). Most likely, the network inversion problem is inherently intractable: The problem it attempts to solve is NP-hard.

Network inversion can also be used in reinforcement-learning problems in a straightforward and elegant way. Given a network model that takes as input the agent's sensations and next action and produces as its output a measure of the quality of that action (e.g., the *dfr* of the agent's next state), gradient *ascent* can be performed in action space to maximize the model's output, thus producing the optimal action for that state [5, 57, 107, 108]. When the *dfr* is used to measure the quality of the agent's chosen action, as was done by Bachrach [5], the result is a *continuous-action* Q-learning algorithm. Of course, the same caveats apply here as apply just above for network inversion in general. If the Q-values are linear with respect to the action space, this problem need not occur since the global minimum of a linear network can be found quickly (cf. Linden [56]). Thrun, *et al.* [108] extended this technique by teaching the actions, once they were determined through gradient descent, to a controller (policy) network which could then suggest these actions quickly and would improve over time such that gradient descent was no longer necessary.

Controller modification techniques, also known as *distal learning* [46], are similar to the method of Thrun, *et al.,* in that a policy or controller module is trained by doing gradient descent in the model. The difference is that these techniques do not use gradient descent to choose actions; they back-propagate the error signal directly into the controller network. The controller produces actions as outputs, and these are fed as the inputs to the model network, as shown in Figure 5.2. When the controller chooses an action for which the

model predicts an undesired result, the difference between the desired and predicted result is back-propagated through the model and into the controller network where the weights are modified to decrease this difference. Future actions by the controller should generate predictions from the model (and therefore reactions from the environment) closer to desired values. This technique has been used both for supervised learning tasks (training the controller to produce specific results in the environment) [44, 46, 69], and for reinforcement-learning tasks (training the controller to maximize the predicted reinforcement) [45, 68, 91, 93, 116]. Several of these designs also include *recurrent* neural networks (see Chapter 4). A taxonomy of many of the different neural-network architectures used for reinforcement learning is given by Lin in his thesis [55], where he describes many different permutations of critics, models, and policy modules.

One of the benefits of using gradient-following methods to choose actions or to modify the controller is that the behavior of the system can be highly tailored simply through modification of the error function. This has been done to model and then to minimize the agent's ignorance of its environment [57], to maximize a balance between the agent's curiosity with and boredom of its environment [91, 93, 108], and to maximize the smoothness, distinctiveness, and speed of a robot's movements [44]. The principal disadvantage of gradient descent techniques is that they are not guaranteed to converge to the optimal policy, unlike table-look-up dynamic programming.

5.5 Some Geometric Intuition

It is common to display the reinforcement-learning process as the construction and utilization of a *reinforcement landscape*. The landscape reflects the "goodness" of each state. The greater a state's elevation on the landscape, the better that state is. The purpose of the critic module or evaluation function is to learn this landscape. Ideally, the landscape should reflect the environment's evaluation function.

Figure 5.3a shows an environment with one reward. In this environment (introduced by Sutton [105]), the agent may move in one of four directions: up, down, left, or right. Assuming the environment is deterministic and the discount factor is 0.9, the resulting landscape is presented in Figure 5.3b. The global maximum is positioned where the single goal is located. (Similar figures are given by Barto, *et al.* [9].)

If the agent has knowledge of its elevation on the landscape, it can immediately judge the quality of its moves and therefore learn the optimal policy. The goodness of a move is determined by the change in elevation plus the immediate reward. If all moves for this state have the same immediate reward, then the greater the agent's increase in elevation, the better the move, and the greater its decrease in elevation, the worse the move. For shortest-path problems, such as that of Figure 5.3, the optimal policy is the steepest path on the surface at each point. Learning this policy means learning to take the action that ascends the surface most quickly.

In general, learning the optimal policy can be done by reinforcing all moves in direct proportion to their immediate reinforcement plus the change they cause in elevation. This is not exactly the learning rule employed by the policy module of the AHC. The learning rule corresponding to the description just given would be:

$$w(s_t, a) \leftarrow w(s_t, a) + \alpha[r(s_t, a) + E(s_{t+1}) - E(s_t)], \tag{5.10}$$

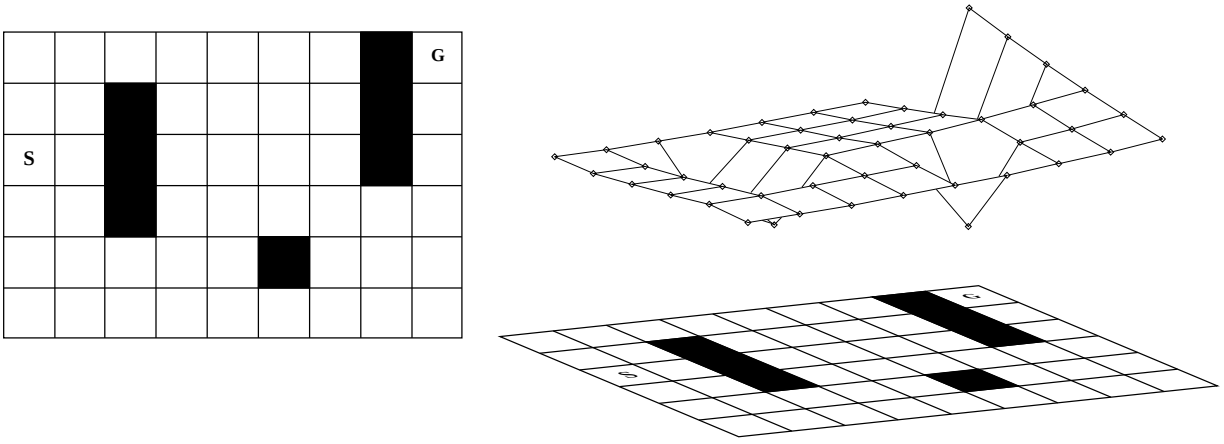

Figure 5.3: *Left (a):* A Markov environment introduced by Sutton [105]. The agent begins in the state labeled "S" and is expected to move to the state labeled "G", where it receives a reward of 1.0. All other transitions receive zero reinforcement. The agent may move up, down, left, or right. *Right (b):* The corresponding landscape. The highest point represents the reinforcement at the goal.

where $E(s_t)$ is the elevation on the landscape of state s_t. If the critic is used as the estimate of a state's elevation, then Equation 5.10 is different from the AHC policy update rule (Equation 5.3) in that there is no discount factor (though the discount factor still appears in the critic update rule, Equation 5.4). In shortest-path problems, both equations lead to the same optimal policies. In the general case, however, since γ in Equation 5.2 regulates the tradeoff between immediate reward and long-term payoff, the two equations may lead to different optimal policies.

Even in shortest-path problems, however, there is another difference: the AHC tends to favor actions in the future that are preferred currently. For example, it is possible as the critic is evolving for its estimate of two states, say x and x_l, to be fairly accurate while the estimate of a third state x_r is far too low. The policy module may then learn to take action l in state x because it leads to state x_l, instead of taking action r which leads to state x_r. Later, the critic's estimate of state x_r's elevation might increase to the same level as x_l's. The agent should learn then that action r is just as good as action l, but it will not. The reason is that on those rare occasions that action r is selected, the agent will arrive in state x_r and see no more improvement in its elevation than it had expected. The weight for action r in state x will not change. The effects of this are not clear, though it could cause state x_r to be underexplored. However, if Equation 5.10 is used, then any time the agent's elevation changes, the weight to the action responsible will be modified. Thus, in the case just given, action r will be reinforced every time it is chosen. This modification has been found to improve performance in at least some cases [53].

The AHC and the method just proposed have a weakness in that weight values reflect both how good an action is as well as how frequently the action was taken. One way to

reduce the effects of update frequency is to remove it completely from the equation. That is, do not make weight *changes* proportional to the difference in elevation, make the *weights* proportional to the difference in elevation:

$$w(s_t, a) \leftarrow w(s_t, a) + \alpha[r(s_t, a) + E(s_{t+1}) - E(s_t) - w(s_t, a)]. \tag{5.11}$$

This equation is extremely similar to that used for Q-learning (Equation 5.6). There is an important difference: Q-values contain absolute as well as relative elevation information. The weights in Equation 5.11 contain only relative elevation information (i.e., the differences in elevation between the current state and those states reachable within a single step). Absolute elevation information is maintained by the elevation estimator, $E(i)$, which is any estimator of each state's "goodness". This is potentially useful for continual learning when the environment or just the reinforcements in the environment change slightly. Often in such cases the absolute elevations may change quite a bit, though little relative information changes (the best action for each state in the new environment may be the same as in the old environment). In these situations it could be quite useful to distinguish between absolute and relative elevation. A similar but more sophisticated method, "Advantage Updating," was introduced by Baird [6]. Advantage updating has been proven to converge to the optimal policy (under appropriate conditions). In simulations it has shown enormous speedups over Q-learning in finely discretized environments. (Though this method was not used in the simulations, it will be mentioned again in Section 8.4).

6

The Automatic Construction of Sensorimotor Hierarchies

The theme of this dissertation is continual learning in reinforcement environments, as discussed in Chapter 1. The purpose of Chapters 2–5 was to provide a foundation for understanding this chapter (and those that follow). This chapter offers the primary technical contribution of the dissertation: a mechanism for addressing the issues of continual learning. The mechanism is the automatic construction of sensorimotor hierarchies. Two approaches to hierarchy construction will be discussed. The aspects of continual learning they address are as follows.

Hierarchical Development. Each hierarchy represents a *skill* or *behavior*, and new behaviors can be constructed from old ones by building onto extant hierarchies. This allows the agent to encapsulate skills from early learning into a foundation for later learning.

Unlimited Behavior Duration. Behaviors are performed over time and carry some degree of state information. Once begun, they generally continue until completion. Because new hierarchies are constructed from existing ones, behaviors that span any arbitrary duration can eventually be built.

Intelligent Behavior Acquisition. The agent builds new hierarchies to represent useful new behaviors. Of the two methods described next, one adds new behaviors to encapsulate sequences of activities the agent has already found to be useful. The other adds hierarchical units when it detects ambiguities that need to be distinguished, thereby allowing the agent to negotiate regions of ambiguous perceptual information.

Incremental Learning. The fabric of the underlying mechanism is a constructive, higher-order neural network that can learn continuously and incrementally. The neural network allows structural credit assignment, and its hierarchical aspects allow "vertical" credit assignment, i.e., assignment of credit to the appropriate level of the hierarchy.

Autonomous Behavior. As with other neural-network methods, those presented here can be combined with reinforcement-learning methods. The resulting agent senses its environment, acts in the environment, and responds to the reinforcement in the environment through temporal credit assignment.

The following sections describe two distinct methods of hierarchy construction. The first (Behavior Hierarchies) is a more intuitive approach; the second (Temporal Transition Hierarchies) while less intuitive, is on the other hand more successful. I will therefore discuss the first in general terms, outlining its intended behavior from a high-level perspective (Section 6.1), while the second I will describe in detail (Section 6.2).

6.1 Behavior Hierarchies

Behavior hierarchies can be described as a system of units in a neural network, where each unit represents a specific behavior sequence. Some units represent primitive behaviors: a single sensation or a single action. Other, "high-level" units represent a sequence of two primitive units, and still higher-level units represent sequences of any two lower units. The system executes a behavior by *choosing* the unit that stands for that behavior. If a primitive action is chosen, the agent executes the action in its environment. If a primitive sensation is chosen, the system determines whether the sensation is present in the environment. If a higher-level unit is chosen, it is decomposed into the two units that it represents; then the first unit's behavior is executed, followed by the second unit's. At any time, a new higher-level skill might be added to the system's abilities by creating a new unit that represents a sequence of two units already in the system.

An example should clarify all this. Suppose the system could sense heat and cold, light and darkness, and that it could move one step north, east, west, or south. Its primitive sensation units would be: SH (sense heat), SC (sense cold), SL (sense light), and SD (sense darkness). Its primitive actions would be: MN (move north), ME (move east), MW (move west), and MS (move south). Now, a new behavior could be created by combining, for example, ME and SC. The new behavior, "Move east and see if it's cold," would be represented by a new unit called: <ME, SC>. After this unit is created, another new unit might be formed, for example: <<ME, SC>, MS> (move east and see if it's cold; if it is, move south). As can be seen from the last example, the rest of a sequence is executed only if the part executed so far has been successful. This allows testing the environment and acting on the result: <SD, MW> (see if it's dark, and if it is, move west).

Behaviors are chosen randomly at first in an effort to achieve a reward. When a reward is received, the system learns that the most recently chosen behaviors may be worth repeating. It is therefore necessary to keep track of the choices made and the level of reinforcement received for these choice sequences. To do this, the entire system is embedded in a neural network, where the connections between units record this information. The stronger the connection from one unit to another, the more likely execution of the first followed by the second will result in reward. Each unit i has two values: $\mathbf{out}^i(t)$, the unit's output value at time t, which is propagated to all other units in the network; and $\mathbf{in}^i(t)$, the network input of the unit as it is received from all other units at time t (just as with the neural networks described in Section 3.3). The output of unit i at time t, $\mathbf{out}^i(t)$, signifies whether or not the behavior represented by unit i has completed: it is 1.0 if the behavior completed at time t, and is 0.0 otherwise.[1] The network input to unit i at time t, $\mathbf{in}^i(t)$, is simply a linear sum of the current output values of all the units in the system:[2]

$$\mathbf{in}^i(t) = \sum_j w_{ij}\mathbf{out}^j(t).$$

As with a standard neural network, w_{ij} is the weight of the connection from j to i. The input value, $\mathbf{in}^i(t)$, of unit i at time t determines the probability that unit i will be chosen

[1] It is possible for more than one behavior to complete at the same time. For example: SC, <ME, SC>, and <MN, <ME, SC>> would all complete whenever <MN, <ME, SC>> completes.

[2] This network is therefore limited to making linear discriminations at each time step. However, by grouping together sensory inputs into high-level behaviors that span multiple time steps, some non-linear discriminations can be made as well.

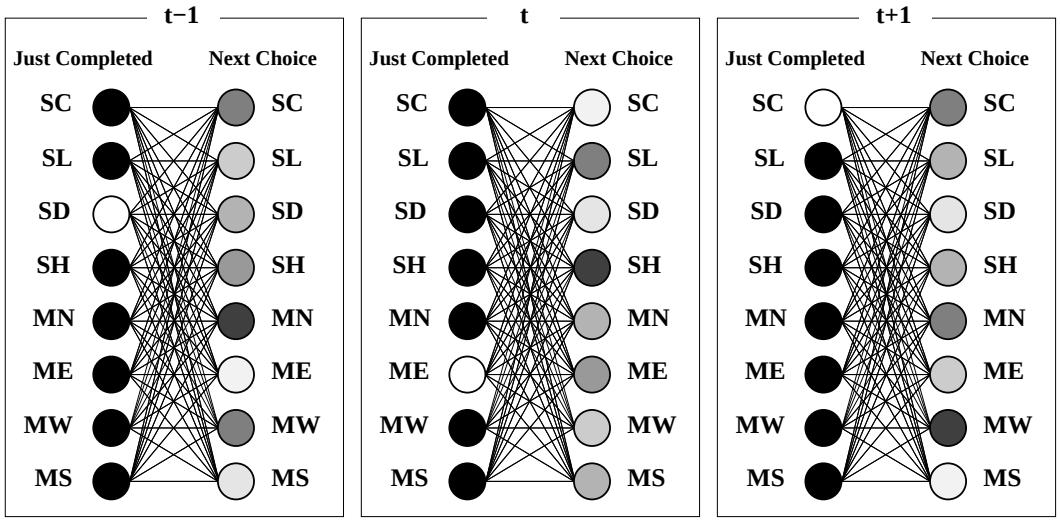

Figure 6.1: A behavior hierarchy network before high-level units are introduced, shown at three successive time steps. The black units are off (have values of 0.0). The white units are on (have values of 1.0). The grey units have various degrees of activation. At time step $t-1$, ME is chosen for execution at step t. At time step t, ME completes, and SC is chosen for execution at step $t+1$. At $t+1$ SC completes (coldness is sensed in the environment).

for execution at time step $t+1$: though stochastic, the unit with the highest input value is the most likely to be chosen.

6.1.1 Network Example

An example network is shown in Figure 6.1 at three time steps: $t-1, t$, and $t+1$. At time step $t-1$, the agent senses darkness only. Through forward propagation many output units are then activated to various degrees. Among these, one is chosen probabilistically: ME. In the following time step, the agent does indeed move east, and the ME unit generates a 1.0 as its output value to the rest of the network. After propagating forward again, another unit is chosen probabilistically: SC, which determines whether coldness is present in the environment. At the following time step, the agent senses coldness and the SC unit generates a 1.0 as output.

Figure 6.2 shows an example of the network with a new, hierarchical unit added in (how new units are added is discussed a little later in Section 6.1.3). In this example when darkness is sensed at time step $t-1$, the <ME, SC> unit is chosen probabilistically. This causes the agent to move east immediately, and then, in the following time step, sensing that the agent has moved east, the SC unit is activated *automatically* (no probabilistic choice is made). At $t+1$, the agent does sense cold, and the <ME, SC> behavior has completed, so both the SC and <ME, SC> units generate output values of 1.0 to be propagated through the network.

Now suppose the unit <<ME, SC>, MS> is built. The resulting network is shown in Figure 6.3. At $t-1$ the <<ME, SC>, MS> unit is chosen (probabilistically). This immediately causes the <ME, SC> unit to be chosen and consequently causes the ME unit to be chosen as well. As a result, time steps t and $t+1$ proceed exactly as in the

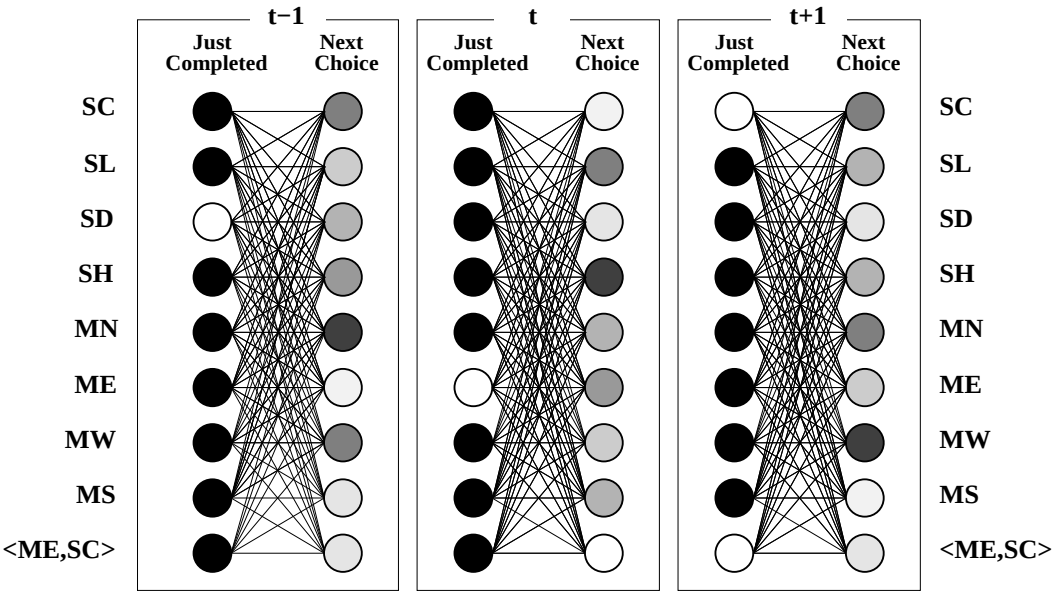

Figure 6.2: A behavior hierarchy network with a single high-level unit, <ME, SC>, shown at three successive time steps. This high-level unit is chosen at time step $t-1$ and consequently causes ME to be chosen at the same time step and SC to be chosen at time step t. Since both ME at t and SC at $t+1$ completed successfully, the output value of $< ME,\ SC >$ is 1.0 at $t+1$. (The unit labels are displayed differently from those in Figure 6.1 for compactness and do not indicate a difference in the network.)

previous paragraph, except that at $t+1$, sensing that <ME,SC> has completed, the network automatically activates the MS unit. At $t+2$, the agent has moved south, and the <<ME, SC>, MS> behavior has completed, so the output value generated by both the MS and the <<ME, SC>, MS> units is 1.0.

6.1.2 Learning

The connection weights are adjusted with the delta rule [118] amplified by the reinforcement signal:

$$\Delta w_{ij}(t) \stackrel{def}{=} \eta R(t+\tau^i)\mathbf{out}^j(t)(T^i(t) - \mathbf{in}^i(t)) \quad (6.1)$$

$$w_{ij}(t) = w_{ij}(t-1) + \Delta w_{ij}(t). \quad (6.2)$$

The weight change at time t of the connection from unit j to unit i is equal to the product of the learning rate η, the reward-level $R(t+\tau^i)$ when unit i's behavior completes, the current output of unit j, and the difference between the activation of unit i and its target $T^i(t)$. The value τ^i represents the number of time steps it takes unit i's behavior to complete: For primitive sense and action units, this value is 1; for hierarchical units, it is the sum of the τ values of the unit's two children. Unit i's target value, $T^i(t)$, is simply the unit's output at its anticipated time of completion, i.e., $T^i(t) = \mathbf{out}^i(t+\tau^i)$.

This rule states that if some unit i is chosen after another unit j's behavior completes, then the system should wait to see if i's behavior completes. If it does not complete (or

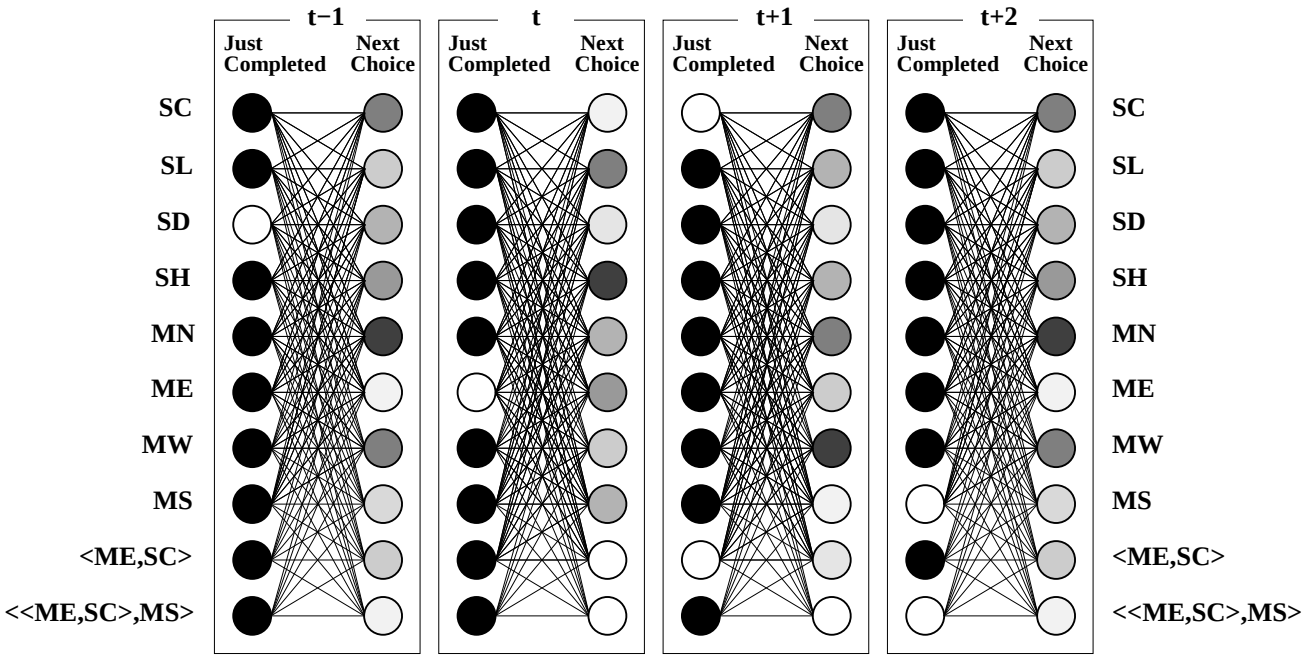

Figure 6.3: The behavior hierarchy network of Figure 6.2 with an additional high-level unit, <<ME, SC>, MS>. The new unit is chosen at time step $t-1$ and immediately causes <ME, SC> to be chosen (thus causing ME also to be chosen at time step $t-1$, followed by SC at time step t). When <ME,SC> completes at time step $t+1$, this automatically activates MS. Both MS and therefore <<ME, SC>, MS> complete at time step $t+2$.

if unit i had not been chosen), then the weight from j to i is decreased by an amount proportional to the current reward and unit i's input. (The system learns not to expect i as much following j; and the more highly unit i was expected — i.e., the greater its input — the greater the resulting change.) But if unit i's behavior *does* complete, then the weight is increased by an amount proportional to the reward received and the amount by which the input to i falls short of its target, 1.0. (The system learns to increase its expectation of i following j; and the smaller the expectation, the greater the weight change.)

6.1.3 An Example of Hierarchy Construction

Figure 6.4 presents an example of how the system could be used. The example is intentionally made simple for clarity. The agent begins at position 1 in the maze. It will receive a reward if it moves to the asterisk in position 6. From position 5, ME should become highly activated because the agent will receive a reward if it moves east. But how can it tell when it is in position 5? It can tell it is in position 5 if it senses light. Therefore, since the sequence SL→ME is always followed by reward, the connection from SL to ME will become strong, and a new unit will be formed, <SL, ME>, to encapsulate this behavior. As units are used in the same sequence again and again, the connections between them get stronger, and other new units will be created such as <SC, <MN, MN>> (useful in position 3), <SH, <MW, MW>> (useful in position 8), and <<<ME, ME>,<SC, <MN, MN>>>, <SL, ME>> (useful in position 1), for example.

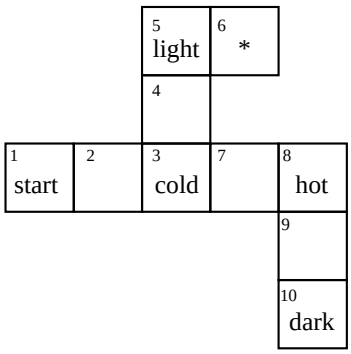

Figure 6.4: An environment for an agent. The agent would start in position 1 and would receive a reward in position 6. Other labels show what sensations the agent would perceive in different parts of the maze.

The units clearly resemble macro-operators, though they have no explicit goals and are designed to be used in reinforcement environments.

6.1.4 Reinforcement Learning with Hierarchies

The learning Equations 6.1 and 6.2 do not take advantage of the hierarchical nature of the system for purposes of reinforcement learning. A modification, however, is straightforward: simply reward the last several *choices* made, and even though the behaviors represented by these choices may span a large period of time, reinforcement is spread smoothly across that time-span. This is achieved by only updating the weights when choices are made, and then slightly modifying Equations 6.1 and 6.2 to be:

$$\begin{aligned} \Delta w_{ij}(t) &\stackrel{def}{=} \mathbf{out}^j(t)(T^i(t) - \mathbf{in}^i(t)) + \sigma \Delta w_{ij}(t-1) \\ w_{ij}(t) &= w_{ij}(t-1) + \eta R(t+\tau^i)\Delta w_{ij}(t), \end{aligned}$$

where $0 \leq \sigma \leq 1$ is a decay parameter that discounts previous weight changes in favor of more recent ones. Each Δw_{ij} is therefore an eligibility trace [8, 50] of weight changes that decays exponentially. The trace constantly accrues weight changes over time — biased towards the most recent ones — but the changes are only applied to the weights when a reinforcement is received. Using an eligibility trace for reinforcement learning, however, is a weak method, and for sequences of many choices compares poorly to the temporal-credit-assignment methods described in Chapter 5.

6.1.5 A Different Approach is Needed

There are problems with the behavior-hierarchy approach. First and most importantly, the behaviors either execute or they do not, and as a result, all units at all levels are binary. This is a discontinuity that prohibits gradient descent and keeps the delta rule from working effectively. Learning tends to be chaotic.

Second, it's possible for multiple behaviors to complete simultaneously, as mentioned above, but it is not possible for them to *begin* simultaneously. For example, if many sensations are impinging on the system at the same time, the sequence in which they are sensed

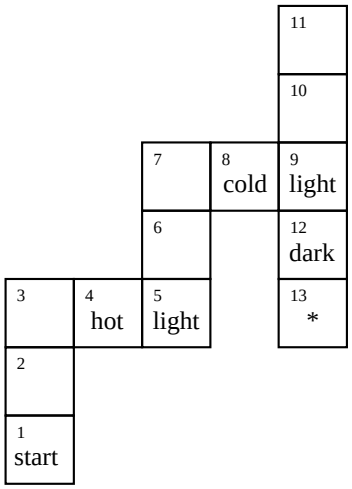

Figure 6.5: In order to decide whether to move north or south in positions 5 and 9, the agent must remember whether it sensed heat or cold in the previous time step.

is irrelevant. In these cases many units could be formed that are functionally identical (grouping together the same sensations), but structurally different (grouping them together in different orders). It should be possible, therefore, for them all to be sensed simultaneously.

Third and finally, there may be many ways of achieving the same end. A behavior that takes the agent from home to work, for example, may meet contingencies along the way. A traffic light may be red when a green light was expected. Yet there is no way to encode this contingency within a single unit such that one action is taken when the light is green, and another is taken when the light is red, both ending with the agent arriving at work. If two behaviors could be chosen at the outset, and in one of these a green light is expected while in the other a red light is expected, then the system would merely be inefficient, requiring enormous numbers of units to encode every possible combination of contingencies. But when only one sequence of behaviors can be chosen at a time, encoding contingencies is not just inefficient; it's impossible. A solution to these problems is described in the next section.

6.2 Temporal Transition Hierarchies

A second method for organizing behaviors hierarchically solves the problems associated with the first, but is less intuitive. It is a *supervised* neural-network algorithm that can be used easily in reinforcement environments by combining it with a reinforcement learning method such as Q-learning (as will be described in Section 7.3). In the *Temporal Transition Hierarchies* network, only the primitive units explicitly represent behaviors. The higher-level units represent *transition strengths* between lower-level units. Instead of combining together behaviors as was done in the last section, these units represent the *degree to which one behavior should follow another* at any particular time. A higher-level unit does this by dynamically modifying the connection weight from one lower-level unit to another.

Figure 6.5 provides an example of how such units could be used. In one case (position 9) the agent should go south when it senses light, while in another case (position 5), it should

go north. To decide whether to move north or south after the light, the agent need only know whether it sensed heat or cold in the previous step. A higher-level unit, <SL, MN>, can be built to strengthen the connection from SL to MN. The system can then learn to activate this unit after sensing heat, but weaken it after sensing cold. (As in Section 6.1, the connection weight from sensory unit j to action unit i indicates the likelihood that the agent will choose action i upon sensing stimulus j. Unlike in Section 6.1, however, sensory units have no input connections, nor do action units have output connections.) Another unit, <SL, MS>, can be built to increase the weight from SL to MS after sensing cold and to decrease it when sensing heat. Now, when heat is sensed in position 4, the transition probability from SL to MN is increased while the transition probability from SL to MS is decreased. The result is that if the agent senses light in the next time step, it will almost certainly move north. The opposite occurs when the agent senses cold in position 8.

6.2.1 Structure and Dynamics

Transition hierarchies are implemented as a constructive, higher-order neural network. The structure of the network can be expressed as follows. Each unit (u^i) in the network is either: a sensory unit ($s^i \in S$); an action unit ($a^i \in A$); or a high-level unit ($l^i_{xy} \in L$) that dynamically modifies w_{xy}, the connection from sensory unit y to action unit x.[3] The action and high-level units can be referred to collectively as non-input units ($n^i \in N$). The next several sections make use of the following definitions:

$$\begin{aligned}
S &\stackrel{def}{=} \{u^i \,|\, 0 \le i < ns\} \\
A &\stackrel{def}{=} \{u^i \,|\, ns \le i < ns + na\} \\
L &\stackrel{def}{=} \{u^i \,|\, ns + na \le i < nu\} \\
N &\stackrel{def}{=} \{u^i \,|\, ns \le i < nu\} \\
u^i(t) &\stackrel{def}{=} \text{the value of the } i\text{th unit at time } t \\
T^i(t) &\stackrel{def}{=} \text{the target value for } a^i(t),
\end{aligned}$$

where ns is the number of sensory units, na is the number of action units, and nu is the total number of units. When it is important to indicate that a unit is a sensory unit, it will be denoted as s^i; similarly, action units will be denoted as a^i, high-level units will be denoted as l^i, and non-input units will be denoted when appropriate as n^i.

The activation of the units is very much like that of a simple, single-layer (no hidden units) network with a linear activation function,

$$n^i(t) = \sum_j \hat{w}_{ij}(t) s^j(t). \tag{6.3}$$

The activation of the i^{th} action or higher-level unit is simply the sum of the sensory inputs multiplied by their respective weights, $\hat{w}_{ij}$, that lead into n^i. The use of a linear activation function and the lack of hidden units would normally spell trouble (see Section 3.1).

[3] A connection may be modified by at most one l unit. Therefore l^i, l^i_{xy}, and l_{xy} are identical but used as appropriate for notational convenience.

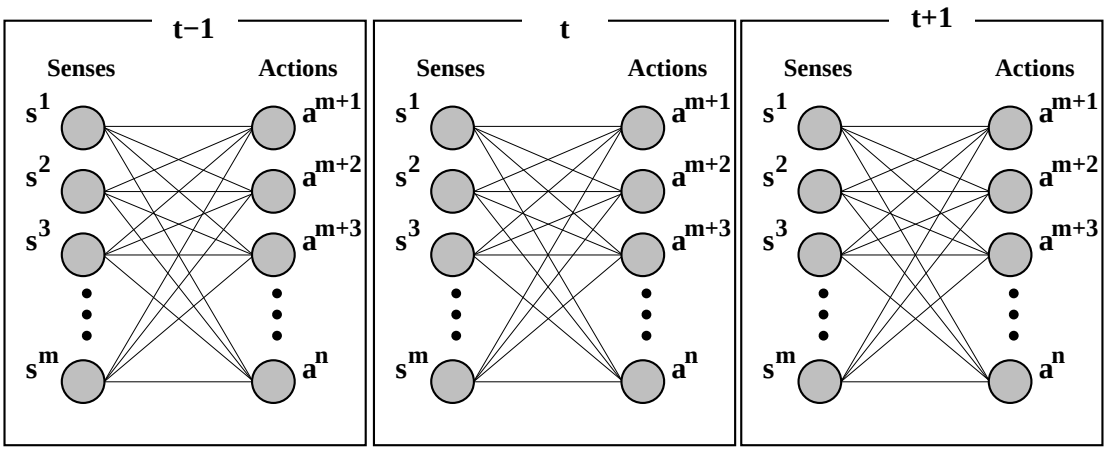

Figure 6.6: A transition hierarchy before high-level units are introduced, shown at three successive time steps (all three are identical). When no high-level units exist, the network functions the same as a simple feed-forward neural network with no hidden units.

However, these are higher-order connections and are therefore capable of non-linear classifications. (More will be said on this in Sections 6.3 and 7.3.5.) The higher-order weights are produced as follows:

$$\hat{w}_{ij}(t) = \begin{cases} w_{ij} + l_{ij}(t-1) & \text{if a high-level unit } l_{ij} \text{ for weight } w_{ij} \text{ exists} \\ w_{ij} & \text{otherwise.} \end{cases} \tag{6.4}$$

If no l unit exists for the connection from i to j, then w_{ij} is used as the weight. If there is such a unit, however, its previous activation value is added to w_{ij} to compute the higher-order weight $\hat{w}_{ij}$.[4]

Figure 6.6 displays an example system at three different time steps. There are m sensory units, $n - m$ action units, and no high-level units. In this case the network behaves as a feed-forward neural-network with no hidden units. In Figure 6.7 there is a single high-level unit, $l^{n+1}_{m+1,3}$. The activation of unit l^{n+1} at $t-1$ is added at time step t to $w_{m+1,3}$ — the weight from unit s^3 to unit a^{m+1} (Equation 6.4). Activation values for the non-input units (a^{m+1} through a^n and l^{n+1}) are then computed using these weights. The new activation of $l^{n+1}(t)$ is then added to $w_{m+1,3}$ at time step $t+1$.

[4]In Section 3.3 it was stated that higher-order connections were multiplicative, whereas the higher-order connections of Equation 6.4 are additive. However, with a bit of mathematical substitution, it can be seen that these qualify as higher-order connections in the usual sense. If one substitutes the right-hand side of Equation 6.3 for l_{ij} in Equation 6.4 (assuming a unit $n^x \equiv l^x_{ij}$ exists) and then replaces $\hat{w}_{ij}$ in Equation 6.3 with the result, then

$$\begin{aligned} n^i(t) &= \sum_j s^j(t)[w_{ij}(t) + \sum_{j'} s^{j'}(t-1)\,\hat{w}_{xj'}(t-1)] \\ &= \sum_j [s^j(t)w_{ij}(t) + \sum_{j'} s^j(t)s^{j'}(t-1)\,\hat{w}_{xj'}(t-1)]. \end{aligned}$$

As a consequence, whenever new units are added, higher orders are introduced while lower orders are preserved.

6.2.2 An Example

Now let us return to Figure 6.5. As described earlier, in position 5 and position 9 the same sense is active, SL, but opposite actions are required. This problem is solved by creating two high-level units $l^9_{\mathrm{MN,SL}}$ and $l^{10}_{\mathrm{MS,SL}}$. (The names SC, SL, SD, SH, MN, ME, MW, and MS are mnemonic equivalents for units 1–4 — the sensory units — and units 5–8 — the action units — respectively.) The resulting network is shown in Figure 6.8. At time step $t-1$ the agent is in position 4 and senses heat. Assuming appropriate connection weights,[5] this causes the action unit ME and the high-level unit $l^9_{\mathrm{MN,SL}}$ to be positively activated and the high-level unit $l^{10}_{\mathrm{MS,SL}}$ to be negatively activated. There are two results. The first and most immediate is that the agent moves east. The second is that at the following time step, t, the weight of the higher-order connection from SL to MN ($\hat{w}_{\mathrm{MN,SL}}$) is greatly increased while the weight from SL to MS ($\hat{w}_{\mathrm{MS,SL}}$) is greatly decreased. Then, since after moving east the agent does sense light at time step t, MN is positively activated and MS is negatively activated.

This method also works for connections into higher-level units. In Figure 6.9, for example, $l^9_{\mathrm{MN,SL}}$ and $l^{10}_{\mathrm{MS,SL}}$ cannot be correctly activated from the sensation of heat or cold alone but require knowledge of what happened one step earlier. Thus, two new units might be built, $l^{11}_{9,\mathrm{SH}}$ and $l^{12}_{10,\mathrm{SH}}$, as shown in Figure 6.10. Assuming appropriate connection weights, the first sets the weight from SH to $l^9_{\mathrm{MN,SL}}$ to a negative value immediately following the sensation of darkness (position 12) and sets it to a high positive value otherwise; the second sets the weight from SH to $l^{10}_{\mathrm{MS,SL}}$ to a high positive value following the sensation of darkness and sets it to a negative value otherwise.

Clearly, this kind of construction can continue indefinitely. A difficulty presents itself, however, if the agent encounters a state with no sensory information (such as position 3 in Figure 6.9). Information is only transmitted over time when higher-level units modify connection weights that affect other units at the following time step. If the agent encounters a state where there is no input, then there will be no values to propagate to any of the units, regardless of the weights. The solution to this problem is to introduce a bias unit as an extra input unit whose value is always 1.0. (The bias unit is not shown in the figures.)

6.2.3 Deriving the Learning Rule

The higher-level units of Section 6.1 were either completely on or completely off, and as such the network could not be differentiated with respect to the error. The transition hierarchy units, however, are continuous, and the network is differentiable. A learning rule can therefore be derived that performs gradient descent in the error space. Since the activations of the l units at one time step are not required until the following time step, all unit-activations can be computed in a single forward-propagation. Gradient descent can also be done much more easily than with recurrent networks (Section 4.3); so, though the derivation that follows is a bit lengthy, the result at the end is a simple learning rule, easy to understand as well as to implement.

[5] The learning method for determining these weights is derived in Section 6.2.3.

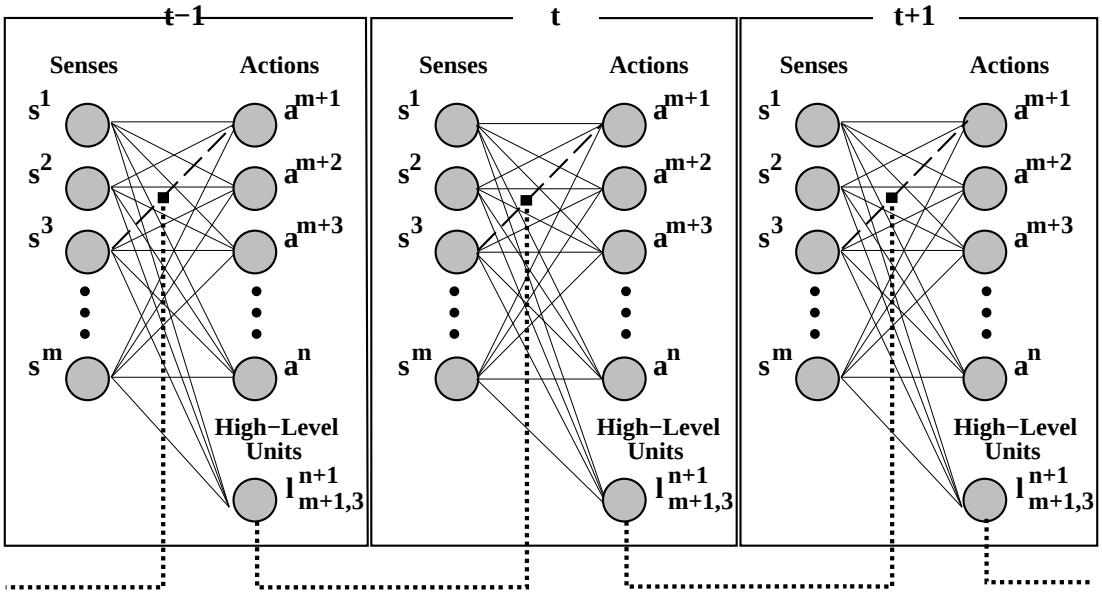

Figure 6.7: A network with a single l unit. This high-level unit, $l^{n+1}_{m+1,3}$, modifies the weight of the connection from unit s^3 to unit a^{m+1}. The dotted line represents the association between $l^{n+1}_{m+1,3}$ at one time step and weight it modifies (the dashed line) at the next.

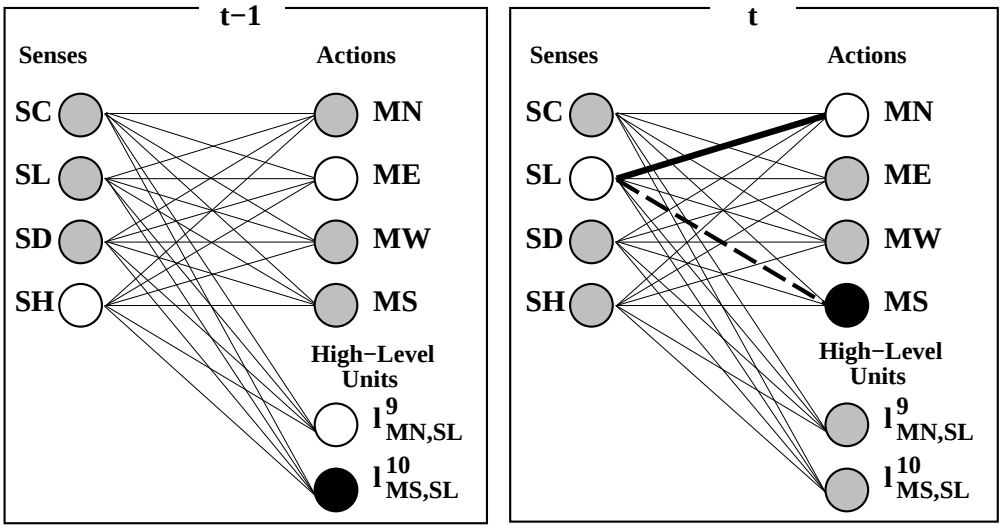

Figure 6.8: This network can solve the task in Figure 6.5. The grey circles denote units with zero activation levels; the white circles represent positive values; and the black circles represent negative values. The dark line represents a strong higher-order connection weight (due to the activation of the weight's higher-order unit at the previous time step), and the dashed line is a negative higher-order weight. At time step $t-1$, the agent is in position 4 and senses heat. The network responds by positively activating the units ME and $l^9_{\mathrm{MN,SL}}$, and by negatively activating unit $l^{10}_{\mathrm{MS,SL}}$. This increases the higher-order weight from SL to MN, and decreases the weight from SL to MS. Because the agent senses light at the next time step, the MN unit is positively activated and the MS unit is negatively activated.

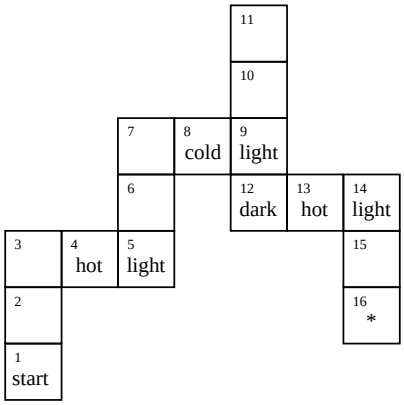

Figure 6.9: In order to decide whether to move north or south in positions 5, 9, and 14, the agent must remember whether it sensed heat or cold in the previous time step. In positions 5 and 14, it must also remember whether it sensed darkness in the time step before that.

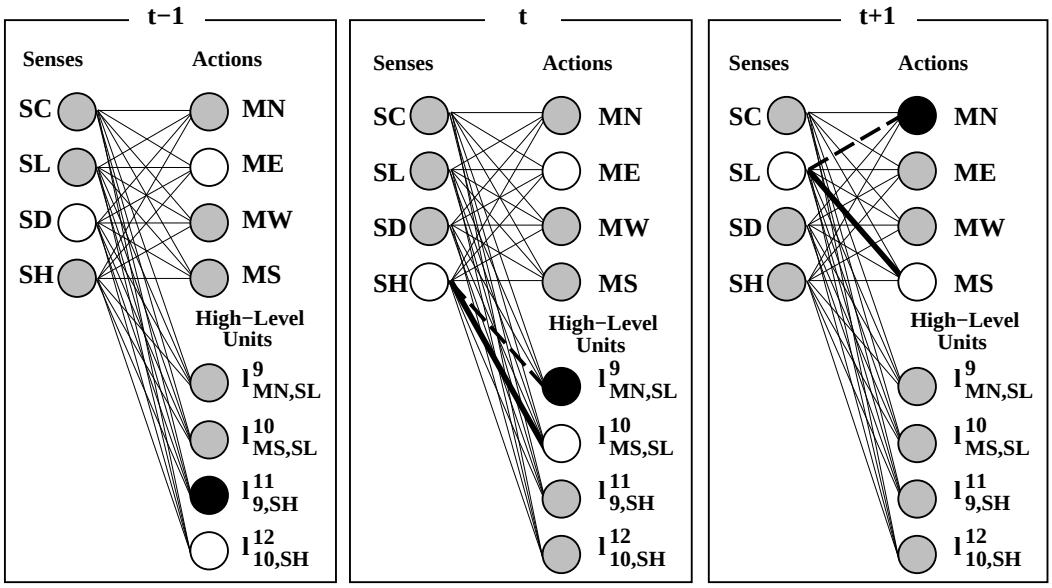

Figure 6.10: This network can solve the task given in Figure 6.9. At time step $t-1$, the agent is in position 12 and senses darkness. This activates the units ME and $l^{12}_{10,\mathrm{SH}}$ (causing the weight from SH to $l^{10}_{\mathrm{MS,SL}}$ to be strongly positive at the next time step), and negatively activates unit $l^{11}_{9,\mathrm{SH}}$ (causing the weight from SH to $l^{9}_{\mathrm{MN,SL}}$ to be negative). At the following time step, t, the agent senses heat, activating ME and $l^{10}_{\mathrm{MS,SL}}$, and negatively activating $l^{9}_{\mathrm{MN,SL}}$. The result at time step $t+1$ is that the agent senses light, and because the weight from SL to MS is strong and the weight from SL to MN is negative, the agent moves south instead of north.

As is common with gradient-descent learning techniques, the network weights are modified so as to reduce the total sum-squared error:

$$\begin{aligned} E &= \sum_t E(t) \\ E(t) &= \frac{1}{2}\sum_i (T^i(t) - a^i(t))^2. \end{aligned} \tag{6.5}$$

In order to allow incremental learning, it is also common to approximate strict gradient-descent by modifying the weights at every time step. At each time step, the weights are changed in the direction opposite their contribution to the error, $E(t)$:

$$\Delta w_{ij}(t) \stackrel{def}{=} \sum_{\tau=0}^{t} \frac{\partial E(t)}{\partial w_{ij}(\tau)} \tag{6.6}$$

$$w_{ij}(t+1) = w_{ij}(t) - \eta \Delta w_{ij}(t)\ , \tag{6.7}$$

where η is the learning rate. The weights, w_{ij}, are time indexed in Equation 6.6 for notational purposes only and are assumed for the purposes of the derivation to remain the same at all time steps (as is done with all incremental neural-network methods).[6]

It can be seen from Equations 6.3 and 6.4 and from Figures 6.8 and 6.10 that it may take multiple time steps for a weight to have an effect on the network's action units. Connections to the action units affect the action units at the current time step. Connections to the first level of high-level units — units that modify connections to the action units — affect the action units after one time step. For example, $\hat{w}_{9,\mathrm{SH}}(t)$, the weight at time step t from SH to $l^9_{\mathrm{MN,SL}}$ in Figure 6.10, affects the action units at time step $t+1$. On the other hand, $\hat{w}_{11,\mathrm{SD}}(t)$, the weight at time step $t-1$ from SD to $l^{11}_{9,\mathrm{SH}}$, affects the action units two time steps later. The "higher" in the hierarchy a unit is, the longer it takes for it (and therefore its incoming connections) to affect the action units. With this in mind, Equation 6.6 can be rewritten as:

$$\Delta w_{ij}(t) \stackrel{def}{=} \frac{\partial E(t)}{\partial w_{ij}(t-\tau^i)}\ , \tag{6.8}$$

where τ^i is the constant value for each action or high-level unit n^i that specifies how many time steps it takes for a change in unit i's activation to affect the network's output. Since this value is directly related to how "high" in the hierarchy unit i is, τ is very easy to compute:

$$\tau^i = \begin{cases} 0 & \text{if } n^i \text{ is an action unit, } a^i \\ 1+\tau^x & \text{if } n^i \text{ is a higher-level unit, } l^i_{xy}. \end{cases} \tag{6.9}$$

The derivation of the gradient proceeds as follows. Define δ^i to be the partial derivative of the error with respect to non-input unit n^i.

$$\delta^i(t) \stackrel{def}{=} \frac{\partial E(t)}{\partial n^i(t-\tau^i)}\ . \tag{6.10}$$

[6]This derivation is done purely within a supervised-learning framework. In contrast, in reinforcement-learning tasks (which will be discussed in Section 7.3), the network's outputs influence what actions the agent chooses and therefore the inputs and target values that the network receives. In this kind of learning, Equation 6.7 really only approximates complete gradient descent.

What must be computed is the partial derivative of the error with respect to each weight in the network:

$$\begin{aligned}\frac{\partial E(t)}{\partial w_{ij}(t-\tau^i)} &= \frac{\partial E(t)}{\partial n^i(t-\tau^i)}\frac{\partial n^i(t-\tau^i)}{\partial w_{ij}(t-\tau^i)} \\ &= \delta^i(t)\frac{\partial n^i(t-\tau^i)}{\partial w_{ij}(t-\tau^i)}\ . \end{aligned} \tag{6.11}$$

From Equations 6.3 and 6.4, the second factor can be derived simply as:

$$\begin{aligned}\frac{\partial n^i(t-\tau^i)}{\partial w_{ij}(t-\tau^i)} &= s^j(t-\tau^i)\frac{\partial \hat{w}_{ij}(t-\tau^i)}{\partial w_{ij}(t-\tau^i)} \\ &= s^j(t-\tau^i)\begin{cases} 1+\frac{\partial l_{ij}(t-\tau^i-1)}{\partial w_{ij}(t-\tau^i)} & \text{if } l_{ij} \text{ exists} \\ 1 & \text{otherwise.}\end{cases}\end{aligned}$$

Because $w_{ij}(t-\tau^i)$ does not contribute to the value of $l_{ij}(t-\tau^i-1)$,

$$\frac{\partial n^i(t-\tau^i)}{\partial w_{ij}(t-\tau^i)} = s^j(t-\tau^i). \tag{6.12}$$

Therefore, combining 6.8, 6.11 and 6.12,

$$\Delta w_{ij}(t) = \frac{\partial E(t)}{\partial w_{ij}(t-\tau^i)} = \delta^i(t)s^j(t-\tau^i)\ . \tag{6.13}$$

Now, $\delta^i(t)$ can be derived as follows. First, there are two cases depending on whether node i is an action unit or a high-level unit:

$$\delta^i(t) = \begin{cases} \dfrac{\partial E(t)}{\partial a^i(t-\tau^i)} & \text{if } n^i \text{ is an action unit, } a^i \\ \dfrac{\partial E(t)}{\partial l^i_{xy}(t-\tau^i)} & \text{if } n^i \text{ is a higher-level unit, } l^i_{xy}. \end{cases} \tag{6.14}$$

The first case is simply the immediate derivative of the error with respect to the action units from Equation 6.5. Since τ^i is zero when n^i is an action unit,

$$\begin{aligned}\frac{\partial E(t)}{\partial a^i(t-\tau^i)} &= \frac{\partial E(t)}{\partial a^i(t)} \\ &= a^i(t) - T^i(t). \end{aligned} \tag{6.15}$$

The second case of Equation 6.14 is somewhat more complicated. First, using Equation 6.4,

$$\frac{\partial E(t)}{\partial l^i_{xy}(t-\tau^i)} = \frac{\partial E(t)}{\partial \hat{w}_{xy}(t-\tau^i+1)}\frac{\partial \hat{w}_{xy}(t-\tau^i+1)}{\partial l^i_{xy}(t-\tau^i)}.$$

Since, from Equation 6.4, $\frac{\partial \hat{w}_{xy}(t)}{\partial l^i_{xy}(t-1)} = 1$,

$$\frac{\partial E(t)}{\partial l^i_{xy}(t-\tau^i)} = \frac{\partial E(t)}{\partial \hat{w}_{xy}(t-\tau^i+1)}.$$

This can now be factored as:

$$\frac{\partial E(t)}{\partial \hat{w}_{xy}(t-\tau^i+1)} = \frac{\partial E(t)}{\partial n^x(t-\tau^i+1)} \frac{\partial n^x(t-\tau^i+1)}{\partial \hat{w}_{xy}(t-\tau^i+1)}.$$

Because n^i is a high-level unit, $\tau^i = \tau^x + 1$ (Equation 6.9). Therefore,

$$\frac{\partial E(t)}{\partial l^i_{xy}(t-\tau^i)} = \frac{\partial E(t)}{\partial n^x(t-\tau^x)} \frac{\partial n^x(t-\tau^x)}{\partial \hat{w}_{xy}(t-\tau^x)},$$

and this can be further reduced using Equations 6.3 and 6.10 as:

$$\frac{\partial E(t)}{\partial l^i_{xy}(t-\tau^i)} = \delta^x(t) s^y(t-\tau^x)$$

Finally, from Equation 6.13,

$$\frac{\partial E(t)}{\partial l^i_{xy}(t-\tau^i)} = \Delta w_{xy}(t). \tag{6.16}$$

Returning now to Equations 6.13 and 6.14, and substituting in Equations 6.15 and 6.16: The change $\Delta w_{ij}(t)$ to the weight w_{ij} from sensory unit s^j to action or high-level unit n^i can be written as:

$$\Delta w_{ij}(t) = \delta^i(t) s^j(t-\tau^i) \tag{6.17}$$

$$= s^j(t-\tau^i) \begin{cases} a^i(t) - T^i(t) & \text{if } n^i \text{ is an action unit, } a^i \\ \Delta w_{xy}(t) & \text{if } n^i \text{ is a higher-level unit, } l^i_{xy}. \end{cases} \tag{6.18}$$

Equation 6.18 is a particularly nice result, since it means that the only values needed to make a weight change at any time step are (1) the error computable at that time step, (2) the input recorded from a specific previous time step, and (3) other weight changes already calculated. This third point is not necessarily obvious; however, each high-level unit is higher in the hierarchy than the units on either side of the weight it affects: $(i > x) \wedge (i > y), \forall l^i_{xy}$. This means that the weights may be modified in a simple bottom-up fashion (described in Section 6.2.5). Error values are first computed for the action units, then weight changes are calculated from the bottom of the hierarchy to the top so that the $\Delta w_{xy}(t)$ in Equation 6.18 will already have been computed before $\Delta w_{ij}(t)$ is computed, for all high-level units l^i_{xy} and all sensory units j.

The intuition behind the learning rule is that each high-level unit, $l^i_{xy}(t)$, learns to utilize the context at time step t to correct its connection's *error*, $\Delta w_{xy}(t+1)$, at time step $t+1$. If the information is available, then the higher-order unit uses it to reduce the error. If the needed information is not available at the previous time step, then new units may be built to look for the information at still earlier time steps.

6.2.4 Adding New Units

New higher-level units are created for reasons different from those of Section 6.1. If one unit is reliably activated after another, there is no reason to interfere with the connection between them. Only when the transition is *unreliable*, that is, when the connection weight

should be different in different circumstances, is a unit required to predict the correct value. (This is reminiscent of Dawkins' "hierarchy of decisions," cf. Section 1.5.) A unit is added whenever a weight is pulled strongly in opposite directions (i.e., when learning is forcing the weight to increase and to decrease at the same time). The unit is created to determine the contexts in which the weight is pulled in each direction.

In order to decide when to add a new unit. Two long-term averages are maintained for every connection. The first of these, $\Delta\bar{w}_{ij}$, is the average change made to the weight. The second, $\Delta\tilde{w}_{ij}$, is the average *magnitude* of the change. When the average change is small but the average magnitude is large, this indicates that the learning algorithm is changing the weight by large amounts but about equally in the positive as in the negative direction; i.e., the connection is being simultaneously forced to increase and to decrease by a large amount.

Two values, Θ and ϵ, are chosen arbitrarily, and when

$$\Delta\tilde{w}_{ij} > \Theta|\Delta\bar{w}_{ij}| + \epsilon, \tag{6.19}$$

that is, when the average magnitude of Δw_{ij} is greater than ϵ more than Θ times the absolute value of the average change, then a new unit is constructed for w_{ij}.

The long-term averages can be computed as follows. The average change is simply

$$\Delta\bar{w}_{ij}(t) = \sigma\Delta w_{ij}(t) + (1-\sigma)\Delta\bar{w}_{ij}(t-1), \quad 0 \leq \sigma \leq 1,$$

where the parameter σ specifies the duration of the long-term average. A smaller value of σ means the average is kept for a longer period of time and is therefore less sensitive to momentary fluctuations. Similarly, the average magnitude of change is given by:

$$\Delta\tilde{w}_{ij}(t) = \sigma|\Delta w_{ij}|(t) + (1-\sigma)\Delta\tilde{w}_{ij}(t-1), \quad 0 \leq \sigma \leq 1.$$

One problem with this method of adding new units, particularly when the weights are changed at every time step, is that certain units may frequently have zero error. This means that their incoming connections would rarely need changes, and this results in long-term averages close to zero. The problem with this is that new units need to be created when a connection is unreliable *in certain contexts*. If the contexts only occur rarely, then it is not possible to determine this unreliability without extremely low σ values. This difficulty is overcome by updating the long-term averages only when changes are actually made to the weight. That is:

$$\Delta\bar{w}_{ij}(t) = \begin{cases} \Delta\bar{w}_{ij}(t-1) & \text{if } \Delta w_{ij}(t) = 0 \\ \sigma\Delta w_{ij}(t) + (1-\sigma)\Delta\bar{w}_{ij}(t-1) & \text{otherwise,} \end{cases} \tag{6.20}$$

and

$$\Delta\tilde{w}_{ij}(t) = \begin{cases} \Delta\tilde{w}_{ij}(t-1) & \text{if } \Delta w_{ij}(t) = 0 \\ \sigma|\Delta w_{ij}|(t) + (1-\sigma)\Delta\tilde{w}_{ij}(t-1) & \text{otherwise.} \end{cases} \tag{6.21}$$

When a new unit is added, its incoming weights are initially zero. It has no output weights: its only task is to anticipate and reduce the error of the weight it modifies. In order to keep the number of new units low, whenever a unit l_{ij}^n is created, the statistics for all connections into the destination unit (u^i) are reset: $\Delta\bar{w}_{ij}(t) \leftarrow -1.0$ and $\Delta\tilde{w}_{ij}(t) \leftarrow 0.0$.

A related method for adding new units, but in feed-forward neural-networks, was introduced by Wynne-Jones [127] and was described briefly in Section 3.2. This method, instead of simply monitoring the statistics for each connection individually, examines *all* the incoming connections to a particular unit to determine whether this group as a whole is being pulled in conflicting directions in the multidimensional weight space. It then creates a new hidden unit to represent an entire area of this multidimensional space. In contrast, new units in the transition hierarchy network learn to correct only a single weight's error.

Even more closely related is Sanger's (also feed-forward) network [89], which occasionally creates a new unit to correct the single weight with the greatest variance. However, unlike temporal transition hierarchies, which can at every time step build units for all weights meeting a specific criterion (Equation 6.19), Sanger's network is trained over a fixed training set until convergence. Only then is a new unit created, and training begins again on the same or a different training set. Its ability to build new units at every time step allows temporal transition hierarchies to learn sequential tasks incrementally and very quickly.

6.2.5 The Algorithm

Because of the simple learning rule and method of adding new units, the learning algorithm is very straightforward. The outline of the procedure is as follows:

For (Ever)
1) Initialize values.
2) Get senses.
3) Propagate Activations.
4) Get Targets.
5) Calculate Weight Changes;
   Change Weights & Weight Statistics;
   Create New Units.

The second and fourth of these are trivial and depend on the task being performed. The first step is simply to make sure all unit values and all delta values are set to zero for the next forward propagation. (The values of the l units at the last time step must, however, be stored for use in step 3.)

1) Initialize values

```
Line    /* Reset all old unit and delta values to zero. */
1.1     For each unit, u(i)
1.2         u(i) ← zero;
1.3         delta(i) ← zero;
```

The third step is nearly the same as the forward propagation in standard feed-forward neural-networks, except for the presence of higher-order units and the absence of hidden layers.

3) Propagate Activations

```
Line    /* Calculate new output values.                                          */
3.1     For each Non-input unit, n(i)
3.2         For each Sensory unit, s(j)

                /* UnitFor(i, j) returns the input of unit l_ij at the last time step.  */
                /* Zero is returned if the unit did not exist. (See Equation 6.4.)      */
3.3             l ← UnitFor(i,j);

                /* To n^i's input, add s^j's value times the (possibly modified)        */
                /* weight from j to i. (See Equation 6.3.)                              */
3.4             n(i) ← n(i) + s(j)*(l + Weight(i, j));
```

The fifth step is the heart of the algorithm. Since the units are arranged as though the input, output, and higher-level units are concatenated into a single vector (i.e., $\forall s^k, a^j, l^i : k < j < i$), whenever a unit l^i_{jk} is added to the network, it is appended to the end of the vector; and therefore $(j < i) \wedge (k < i)$. This means that when updating the weights, the δ^i's and Δw_{ij}'s of Equation 6.18 must be computed with i in ascending order, so that Δw_{xy} will be computed before any Δw_{ij} for unit l^i_{xy} is computed.

If a weight change is not zero, it is applied to the weight. If the weight has no higher-level unit, the weight statistics are updated and checked to see whether a higher-level unit is warranted. If a unit is warranted for the weight leading from unit j to unit i, a unit is built for it, and the statistics are reset for all weights leading into unit i. If a higher-level unit already exists, that unit's delta value is calculated.

While testing the algorithm, it became apparent that changing the weights at the bottom of a large hierarchy could have an explosive effect: the weights would oscillate to ever larger values. This indicated that a much smaller learning rate was needed for these weights. Two learning rates were therefore introduced: the normal learning rate, η, for weights without higher-level units (i.e., $w_{xy}|\neg\exists l^i_{xy}$); and a fraction, η_L, of η for those weights whose values *are* affected by higher-level units, (i.e., $w_{xy}|\exists l^i_{xy}$).

5) Update Weights and Weight Statistics; Create New Units.

```
Line   /* Calculate δ_i for the action units, a^i. (See Equation 6.18.)                */
5.1    For each action unit, a(i)
5.2       delta(i) = a(i) - Target(i);

       /* Calculate all Δw_ij's, Δw̄_ij's, Δw̃_ij's.                                    */
       /* For higher-order units l^i, calculate δ^i's.                                */
       /* Change weights and create new units when needed.                            */
5.3    For each Non-input unit, n(i), with i in ascending order
5.4       For each Sensory unit, s(j)

             /* Compute weight change (Equation 6.17).                                */
             /* Previous(j, i) retrieves s^j(t - τ^i).                                */
5.5          delta_w(i, j) ← delta(i) * Previous(j, i);

             /* If Δw_ij ≠ 0, update weight and statistics. (Eqs. 6.20 and 6.21).     *
5.6          if (delta_w(i,j) ≠ 0)

                /* IndexOfUnitFor(i, j) returns n for l^n_ij; or -1 if l^n_ij does not exist. */
5.7             n ← IndexOfUnitFor(i, j);

                /* If l^n_ij doesn't exist: update statistics, learning rate is η.   */
5.8             if (n = -1)

                   /* Change weight w_ij. (See Equation 6.7.)                        */
5.9                Weight(i, j) ← Weight(i, j) - ETA * delta_w(i, j);

                   /* Update long-term average, Δw̄_ij. (See Equation 6.20)           */
5.10               lta(i,j) ← SIGMA * delta_w(i,j) + (1-SIGMA) * lta(i,j);

                   /* Update long-term mean absolute deviation Δw̃_ij. (Eq. 6.21)     */
5.11               ltmad(i,j) ← SIGMA * abs(delta_w(i,j)) +
                                (1-SIGMA) * ltmad(i,j);

                   /* If Higher-Order unit l^n_ij should be created (Equation 6.19) ... */
5.12               if (ltmad(i, j) > THETA * abs(lta(i, j)) + EPSILON)

                      /* ... create unit l^N_ij, (where N is the current network size). */
5.13                  BuildUnitFor(i, j);

                      /* Reset statistics for all incoming weights.                  */
5.14                  For each Sensory unit, s(k)
5.15                     lta(i, k) ← -1.0;
5.16                     ltmad(i, k) ← 0.0;

                /* If l^n_ij does exist (n≠-1), store δ^n (Equation 6.14 and 6.16).  */
                /* Change w_ij, learning rate = η_L * η.                             */
5.17            else
5.18               delta(n) ← delta_w(i, j);
5.19               Weight(i, j) ← Weight(i, j) - L_ETA*ETA * delta_w(i, j);
```

6.2.6 Tracing Through the Algorithm

It may be helpful to trace through the algorithm with a simple example. This potentially tedious exercise is done here with about the simplest interesting example: sequential equivalence (this is the same as XOR, only the opposite). In this supervised learning task, there are two input units: s^1 and s^2. Exactly one of these has a value of 1.0 at each time step; the other has a value of 0.0. There is a single output unit, a^3. The target at each time step is supplied by a teacher. The target is 1.0 if the input at the current and previous time steps were the same (i.e., if $s^1(t) = s^1(t-1)$ and therefore $s^2(t) = s^2(t-1)$). The target is 0.0 if the current and previous inputs were different (i.e., if $s^1(t) \neq s^1(t-1)$ and therefore $s^2(t) \neq s^2(t-1)$).

The following example uses a very high learning rate so as to accomplish the most possible in as few pages as possible. It also uses a large σ and small Θ and ϵ values so as to create units as quickly as possible (for the same reason). These are the parameters used:

$$\begin{aligned} \eta &= 0.5 \\ \eta_L &= 1.0 \\ \sigma &= 0.3 \\ \theta &= 0.4 \\ \epsilon &= 0.1 \end{aligned}$$

Table 6.1 traces through the training algorithm over the first five time steps. The target for the first time step is undefined, because there was no previous input to compare the current input against. The following labeled explanations describe the highlights of the learning process and correspond to the labeled entries in Table 6.1. References to the respective line numbers of the algorithm are also given when possible.

Starting with time step 2 (Second column of Table 6.1):

A. Since all weights are initially zero, the output of the network is 0.0 (Line 3.4).
B. The target is 1.0 because the input at time step 1 is the same as the input at time step 2.
C. The error value for a^3 is the output minus the target (Line 5.2).
D. $s^1(t - \tau^3) = s^1(2) = 0.0$, so $\Delta w_{3,1} = 0.0 * \delta^3 = 0.0$ (Line 5.5).
E. $s^2(t - \tau^3) = s^2(2) = 1.0$, so $\Delta w_{3,2} = 1.0 * \delta^3 = -1.0$ (Line 5.5).
F. The value $\eta \Delta w_{3,2} = -0.5$ is subtracted from the weight (Line 5.9).
G. Since $\Delta \bar{w}_{3,2}$ was already -1.0, averaging in δ^3 does not change it (Line 5.10).
H. $\Delta \tilde{w}_{3,2} \Leftarrow (1 - \sigma)\Delta \tilde{w}_{3,2} + \sigma |\Delta w_{3,1}| = .7 * 0.0 + .3 * 1.0 = 0.3$ (Line 5.11).
I. Since $\Delta \tilde{w}_{3,2}$ is not greater than this value, don't build a new unit (Line 5.12).

Time step 3 (third column of Table 6.1):

A. The output is 0.0 again since one weight is zero and the other is multiplied by s^2, which is zero (Line 3.4).
B. The target is 0.0 because the current input is different from the input at the last time step.
C. Since the target and output matched, there is no error (Line 5.2).

Time Step:	1	2	3	4	5
2) Get Senses.					
s^1:	0	0	1	1	0
s^2:	1	1	0	0	1
3) Propagate Activations.					
a^3:	0.00	0.00 **A**	0.00 **A**	0.00 **A**	0.50 **A**
4) Get Targets.					
T^3:	0.00	1.00 **B**	0.00 **B**	1.00	0.00
5) Update Weights, etc.					
δ^3:	0.00	-1.00 **C**	0.00 **C**	-1.00 **B**	0.50
$\Delta w_{3,1}$:	0.00	0.00 **D**	0.00	-1.00 **C**	0.00
$w_{3,1}$:	0.00	0.00	0.00	0.50	0.50
$\Delta \bar{w}_{3,1}$:	-1.00	-1.00	-1.00	-1.00	-1.00
$\Delta \tilde{w}_{3,1}$:	0.00	0.00	0.00	0.30	0.30
$\Theta\lvert\Delta \bar{w}_{3,1}\rvert + \epsilon$:	0.50	0.50	0.50	0.50	0.50
$\Delta w_{3,2}$:	0.00	-1.00 **E**	0.00	0.00	0.50 **B**
$w_{3,2}$:	0.00	0.50 **F**	0.50	0.50	0.25
$\Delta \bar{w}_{3,2}$:	-1.00	-1.00 **G**	-1.00	-1.00	-0.55 **C**
$\Delta \tilde{w}_{3,2}$:	0.00	0.30 **H**	0.30	0.30	0.36 **D**
$\Theta\lvert\Delta \bar{w}_{3,2}\rvert + \epsilon$:	0.50	0.50 **I**	0.50	0.50	0.32 **E**

Table 6.1: The first five time steps of learning the equivalence function. The time steps progress from left to right. Stages 2–5 of the learning algorithm progress from top to bottom at each time step. s^1 and s^2 are the inputs to the network. a^3 is the network's output. T^3 is the target value (for a^3). δ^3 is the error value for a^3. Similarly, the remaining rows are labeled according to the notation presented in the previous sections. The boldface capital letters next to some entries refer to corresponding explanations in the text.

Time step 4:

A. The value of the output unit is still zero. (A weight's value at the current time step is the value *before* the weight is changed, and is actually displayed in the previous column.)
B. The difference between output and target is again -1.0 (Line 5.2).
C. This time, however, s^1 is 1.0, so $w_{3,1}$ is changed (Line 5.5).

Time step 5:

A. The output is: $s^1 * w_{3,1} + s^2 w_{3,2} = 0.0 * 0.5 + 1.0 * 0.5 = 0.5$ (Line 3.4).
B. $s^2(t - \tau^3)\delta^3 = s^2(5)\delta^3 = 1.0 * 0.5 = 0.5$ (Line 5.5).
C. $(1 - \sigma)\Delta \bar{w}_{3,2} + \sigma \Delta w_{3,2} = 0.7 * -1.0 + 0.3 * 0.5 = -0.55$ (Line 5.10).
D. $(1 - \sigma)\Delta \tilde{w}_{3,2} + \sigma |\Delta w_{3,2}| = 0.7 * 0.3 + 0.3 * |0.5| = 0.36$ (Line 5.11).
E. $\Delta \tilde{w}_{3,2}$ is greater than this value, 0.32, which means that a new unit, $l^4_{3,2}$, will be created for this connection (Line 5.12 and 5.13).

Time Step:	6	7	8	9	10	11
2) Get Senses.						
s^1:	0	1	1	0	0	1
s^2:	1	0	0	1	1	0
3) Propagate Activations.						
a^3:	0.25 **A**	0.50 **A**	0.25 **A**	0.62 **A**	0.69 **A**	0.62
l^4:	0.00 **B**	0.00	0.00	0.38 **B**	0.38	-0.31
4) Get Targets.						
T^3:	1.00	0.00	1.00	0.00	1.00	0.00
5) Update Weights, etc.						
δ^3:	-0.75 **C**	0.50	-0.75	0.62	-0.31	0.62
$\Delta w_{3,1}$:	0.00	0.50	-0.75	0.00	0.00	0.62
$w_{3,1}$:	0.50	0.25	0.62	0.62	0.62	0.31
$\Delta \bar{w}_{3,1}$:	-1.00 **D**	-0.55	-0.61	-0.61	-0.61	-0.24
$\Delta \tilde{w}_{3,1}$:	0.00 **D**	0.15	0.33	0.33	0.33	0.42 **A**
$\Theta\|\Delta \bar{w}_{3,1}\| + \epsilon$:	0.50	0.32	0.34	0.34	0.34	0.20 **A**
$\Delta w_{3,2}$:	-0.75 **E**	0.00	0.00	0.62	-0.31	0.00
$w_{3,2}$:	0.62 **F**	0.62	0.62	0.31	0.47	0.47
$\Delta \bar{w}_{3,2}$:	-1.00 **G**	-1.00	-1.00	-1.00	-1.00	-1.00
$\Delta \tilde{w}_{3,2}$:	0.00 **G**	0.00	0.00	0.00	0.00	0.00
$\Theta\|\Delta \bar{w}_{3,2}\| + \epsilon$:	0.50	0.50	0.50	0.50	0.50	0.50
$l_{3,2}$:	4 **H**	4	4	4	4	4
δ^4:	-0.75 **I**	0.00	0.00	0.62	-0.31	0.00
τ^4:	1 **J**	1	1	1	1	1
$\Delta w_{4,1}$:	0.00 **K**	0.00	0.00	0.62 **C**	0.00	0.00
$w_{4,1}$:	0.00 **K**	0.00	0.00	-0.31 **C**	-0.31	-0.31
$\Delta \bar{w}_{4,1}$:	-1.00	-1.00	-1.00	-0.51	-0.51	-0.51
$\Delta \tilde{w}_{4,1}$:	0.00	0.00	0.00	0.19	0.19	0.19
$\Theta\|\Delta \bar{w}_{4,1}\| + \epsilon$:	0.50	0.50	0.50	0.30	0.30	0.30
$\Delta w_{4,2}$:	-0.75 **K**	0.00	0.00	0.00	-0.31	0.00
$w_{4,2}$:	0.38 **K**	0.38	0.38	0.38	0.53	0.53
$\Delta \bar{w}_{4,2}$:	-0.92	-0.92	-0.92	-0.92	-0.74	-0.74
$\Delta \tilde{w}_{4,2}$:	0.22	0.22	0.22	0.22	0.25	0.25
$\Theta\|\Delta \bar{w}_{4,2}\| + \epsilon$:	0.47	0.47	0.47	0.47	0.40	0.40

Table 6.2: Time steps 6–11 while learning the equivalence function. A new high-level unit, $l^4_{3,2}$, has been added to the network together with its two incoming connections. Again, the capital letters next to some entries refer to the corresponding explanations in the text.

Time step 6 (shown in Table 6.2 where the new unit, $l^4_{3,2}$ can now be seen):

A. Since $l^4_{3,2}$ did not exist at the previous time step, it has no effect on a^3 at this time step (Line 3.4).
B. The new unit's value is zero because the initial weights into a unit are zero (Line 5.13).
C. The error for a^3 is computed as before: output minus target (Line 5.2).
D. These values were reinitialized when $l^4_{3,2}$ was created (Lines 5.15, and 5.16).
E. Though $w_{3,2}$ now has a high-level unit associated with it, its delta value is computed as before (Line 5.5).
F. The weight is changed slightly differently, though. The learning rate is now $\eta_L * \eta$. In this example, however, because $\eta_L = 1.0$, there is no resulting difference (Line 5.19).
G. Since the weight already has a high-level unit, its statistics are no longer updated (Line 5.17).
H. This merely displays the index of the high-level unit associated with $w_{3,2}$. This is not a changeable value. Once a high-level unit is created for a weight, the association between them is permanent (Line 5.13).
I. The error value for a high-level unit is the same as the error value of the weight the unit modifies, $\Delta w_{3,2}$ (Line 5.18).
J. Because unit 4 is a high-level unit, it has a non-zero τ value. The weight that unit $l^4_{3,2}$ modifies, $w_{3,2}$, feeds into an output unit, a^3 (which has a τ value of 0). Therefore $\tau^4 = 1 + \tau^3 = 1$. This value is fixed once the unit is created (Line 5.13).
K. The weights into $l^4_{3,2}$ are changed the same as any other weight, except that the Δw values are computed using the input values at time step $t - \tau^4$ (i.e., time step 5). The learning rate is η, since there is no high-level unit $l_{4,2}$ associated with this weight (Lines 5.5, 5.6, and 5.9).

Time steps 7 and 8:

A. Since $l^4_{3,2} = 0.0$ at the previous time step, it has no effect on a^3 (Line 3.4).

Time step 9:

A. High-level unit $l^4_{3,2}$ was again 0.0 at the previous time step, so it still has no effect on a^3 (Line 3.4).
B. However, $l^4_{3,2}$ finally has a non-zero value at the current time step (Line 3.4). ($s^1 w_{4,1} + s^2 w_{4,2} = 0.0 * 0.0 + 1.0 * 0.38 = 0.38$).
C. Weight $w_{4,1}$ is changed because the input from s^1 was non-zero at time step $t - \tau^4$ (i.e., time step 8). $\Delta w_{4,1} = s^1(t - \tau^4)\delta^4 = s^1(8)\delta^4 = 1.0 * 0.62$ (Line 5.5).

Time step 10:

A. Because (finally) $l^4_{3,2}$ was not 0.0 at the previous time step, it temporarily modifies $w_{3,2}$ and therefore has an effect on a^3. As a result, $a^3 = s^1 w_{3,1} + s^2(w_{3,2} + l^4_{3,2}) = 0.0 * 0.62 + 1.0(0.31 + 0.38) = 0.69$ (Lines 3.3 and 3.4).

Time step 11:

A. $\Delta\tilde{w}_{3,1} > \Theta|\Delta\bar{w}_{3,1}| + \epsilon$, so a new unit is created, $l^5_{3,1}$ (Lines 5.12 and 5.13).

Time Step:	12	21	22	23	24	25
2) Get Senses.						
s^1:	1	0	0	1	1	0
s^2:	0	1	1	0	0	1
3) Propagate Activations.						
a^3:	0.31 **A**	0.01	1.00 **A**	0.04 **A**	0.98 **A**	0.00 **A**
l^4:	-0.31	0.58 **A**	0.58 **B**	-0.42 **B**	-0.42	0.58
l^5:	0.00	-0.41 **A**	-0.41 **C**	0.55 **C**	0.55	-0.43
4) Get Targets.						
T^3:	1.00	0.00	1.00	0.00	1.00	0.00
5) Update Weights, etc.						
δ^3:	-0.69 **B**	0.01	-0.00	0.04	-0.02	0.00
$\Delta w_{3,1}$:	-0.69 **C**	0.00	0.00	0.04	-0.02	0.00
$w_{3,1}$:	0.66 **C**	0.45	0.45	0.43	0.44	0.44
$\Delta \bar{w}_{3,1}$:	-1.00 **D**	-1.00	-1.00	-1.00	-1.00	-1.00
$\Delta \tilde{w}_{3,1}$:	0.00 **D**	0.00	0.00	0.00	0.00	0.00
$\Theta\|\Delta \bar{w}_{3,1}\| + \epsilon$:	0.50	0.50	0.50	0.50	0.50	0.50
$l_{3,1}$:	5 **E**	5	5	5	5	5
$\Delta w_{3,2}$:	0.00	0.01	-0.00	0.00	0.00	0.00
$w_{3,2}$:	0.47	0.42	0.42	0.42	0.42	0.42
$\Delta \bar{w}_{3,2}$:	-1.00	-1.00	-1.00	-1.00	-1.00	-1.00
$\Delta \tilde{w}_{3,2}$:	0.00	0.00	0.00	0.00	0.00	0.00
$\Theta\|\Delta \bar{w}_{3,2}\| + \epsilon$:	0.50	0.50	0.50	0.50	0.50	0.50
$l_{3,2}$:	4	4	4	4	4	4
δ^4:	0.00	0.01	-0.00	0.00	0.00	0.00
τ^4:	1	1	1	1	1	1
$\Delta w_{4,1}$:	0.00	0.01	0.00	0.00	0.00	0.00
$w_{4,1}$:	-0.31	-0.42	-0.42	-0.42	-0.42	-0.42
$\Delta \bar{w}_{4,1}$:	-0.51	-0.14	-0.14	-0.14	-0.14	-0.10
$\Delta \tilde{w}_{4,1}$:	0.19	0.10	0.10	0.10	0.10	0.07
$\Theta\|\Delta \bar{w}_{4,1}\| + \epsilon$:	0.30	0.16	0.16	0.16	0.16	0.14
$\Delta w_{4,2}$:	0.00	0.00	-0.00	0.00	0.00	0.00
$w_{4,2}$:	0.53	0.58	0.58	0.58	0.58	0.58
$\Delta \bar{w}_{4,2}$:	-0.74	-0.39	-0.27	-0.27	-0.27	-0.27
$\Delta \tilde{w}_{4,2}$:	0.25	0.15	0.10	0.10	0.10	0.10
$\Theta\|\Delta \bar{w}_{4,2}\| + \epsilon$:	0.40	0.25	0.21	0.21	0.21	0.21
δ^5:	-0.69 **F**	0.00	0.00	0.04	-0.02	0.00
τ^5:	1 **G**	1	1	1	1	1
$\Delta w_{5,1}$:	-0.69 **H**	0.00	0.00	0.00	-0.02	0.00
$w_{5,1}$:	0.34 **H**	0.55	0.55	0.55	0.56	0.56
$\Delta \bar{w}_{5,1}$:	-0.91	-0.54	-0.54	-0.54	-0.38	-0.38
$\Delta \tilde{w}_{5,1}$:	0.21	0.19	0.19	0.19	0.14	0.14
$\Theta\|\Delta \bar{w}_{5,1}\| + \epsilon$:	0.46	0.32	0.32	0.32	0.25	0.25
$\Delta w_{5,2}$:	0.00	0.00	0.00	0.04	0.00	0.00
$w_{5,2}$:	0.00	-0.41	-0.41	-0.43	-0.43	-0.43
$\Delta \bar{w}_{5,2}$:	-1.00	-0.30	-0.30	-0.20	-0.20	-0.20
$\Delta \tilde{w}_{5,2}$:	0.00	0.19	0.19	0.14	0.14	0.14
$\Theta\|\Delta \bar{w}_{5,2}\| + \epsilon$:	0.50	0.22	0.22	0.18	0.18	0.18

Table 6.3: A new high-level unit, $l^5_{3,1}$, has been added to the network. This unit allows the equivalence function to be learned after some further training (steps 13–20, not shown). By time step 21, the network's output is nearly perfect. See text for information about entries labeled with capital letters.

Because of the new unit and its new weights, the remainder of the example is carried out in Table 6.3. **Time step 12**:

A. The output is computed as in the previous steps — unit $l^5_{3,1}$ has no effect yet (Line 3.4).
B. The output unit has an error value as before (Line 5.2).
C. This connection has a high-level unit, so the learning rate is $\eta_L * \eta$ (Line 5.5 and 5.19).
D. The statistics for the connection were reset in the last time step (Lines 5.15 and 5.16).
E. The index of the high-level unit modifying this connection ($l^5_{3,1}$) is 5 (Line 5.7).
F. The error value for the new high-level unit is the same as that of the weight it modifies, $\Delta w_{3,1}$ (Line 5.18).
G. $\tau^3 = 0$, so $\tau^5 = 1 + \tau^3 = 1$.
H. Because $\tau^5 = 1$, the input values used to compute the weight changes at time step 12 come from time step 11 ($t - \tau^5 = 12 - 1 = 11$): $\Delta w_{5,1} = \delta^5 * s^1(11) = 0.69 * 1.0$.

Though not shown in Table 6.3, training continues to modify the connection weights. At time step 21, the network outputs are nearly perfect (no additional units are created). Time steps 22–25 show all four possible cases for the equivalence function.

Time step 21:

A. The value of unit $l^4_{3,2}$ is 0.58, and the value of unit $l^4_{3,1}$ is -0.41. This indicates that at the following time step, if unit s^2 is activated, then output unit a^3 should become strongly activated, but if instead input unit s^1 is activated, then a^3 should be weakly activated.

Time step 22:

A. The value of the output unit at the current time step is:
$$\begin{aligned} a^3(22) &= s^1(22)[w_{3,1}(22) + l^5_{3,1}(21)] + s^2(22)[w_{3,2}(22) + l^4_{3,2}(21)] \\ &= 0.0(0.45 - 0.41) + 1.0(0.42 + 0.58) \\ &= 1.0. \end{aligned}$$
B. $s^1 w_{4,1} + s^2 w_{4,2} = 0.0 * -0.42 + 1.0 * 0.58$.
C. $s^1 w_{5,1} + s^2 w_{5,2} = 0.0 * 0.55 + 1.0 * -0.41$.

Time step 23:

A. $$\begin{aligned} a^3(23) &= s^1(23)[w_{3,1}(23) + l^5_{3,1}(22)] + s^2(23)[w_{3,2}(23) + l^4_{3,2}(22)] \\ &= 1.0(0.45 - 0.41) + 0.0(0.42 + 0.58) \\ &= 0.04. \end{aligned}$$
B. $s^1 w_{4,1} + s^2 w_{4,2} = 1.0 * -0.42 + 0.0 * 0.58 = -0.42$.
C. $s^1 w_{5,1} + s^2 w_{5,2} = 1.0 * 0.55 + 0.0 * -0.41 = 0.55$.

Time step 24:

A. $1.0(0.43 + 0.55) + 0.0(0.42 - 0.42) = 0.98$.

Time step 25:

A. $0.0(0.44 + 0.55) + 1.0(0.42 - 0.42) = 0.0$.

6.3 Conclusions

Two important characteristics distinguish transition hierarchies from behavior hierarchies. First, transition hierarchies have no discontinuities. The activation function of the action and higher-level units is continuous (in fact, it's linear[7]), meaning these units do not need to be completely on or completely off. This not only makes it possible to do gradient descent, but means that behaviors may be active to varying *degrees*. Because of this and because high-level units in a transition hierarchy initially have no effect but only over time begin to reduce error, the learning curve tends to decrease fairly monotonically, as will be seen in Section 7.2.

Second, an agent that uses temporal transition hierarchies to choose actions implements a system of *distributed hierarchical control*; instead of a single unit encoding an entire behavior, it is the pattern of activations across input and high-level units that determines the behavior to be performed. The unit activations represent a set of context-sensitive plans: sequential plans embedded with contingency information. Each high-level unit represents a particular modification to make in the network given specific expected and unexpected input events. If the event is expected, the high-level unit can be viewed as a link in a plan, carrying from a previous time step to the next the agent's intention to execute this plan. If the event is not expected, the unit can be thought of as providing contingencies for the plan.

For example, let's say the desired behavior is: move east, and if it's cold, move south, but if it's hot, move north. This can be done by activating *all* of the following units: ME, $l_{\mathrm{MS,SC}}$, and $l_{\mathrm{MN,SH}}$ (while negatively activating any conflicting units). The behavior hierarchies of Section 6.1 encountered problems because only one behavior could be active at a time. These problems disappear completely in the new system since multiple high-level units can be activated simultaneously.

Transition hierarchies may also be viewed as a system of continuous-valued condition-action rules that are inserted or removed depending on another set of such rules that are in turn inserted or removed depending on another set, etc. When new rules (new units) are added, they are initially invisible to the system, (i.e., they have no effect), but only gradually learn to have an effect as the opportunity to decrease error presents itself.

"Encapsulation" of a behavior, an important concept to the behavior hierarchies of Section 6.1 still occurs, but differently. In Temporal Transition Hierarchies, if the weight from a sense to an action is strong, then the sense→action behavior is effectively encapsulated. There might be a whole stream of such behaviors that are essentially reflexive; once the stimulus is perceived, the action occurs. The boundaries between encapsulated behaviors appear when an immediate response to a stimulus is not known. This is where a new unit is created to find the context that will determine the correct response. Once the context is found, the new unit binds together two streams of reflexive responses into a single stream of responses, thus constituting a newly encapsulated, context-sensitive behavior.

Limitations. One problem with Temporal Transition Hierarchies is that there are no hidden units in the traditional sense, and the activation functions are linear. Linear activation

[7]It can still make non-linear discriminations, however, due to its use of higher-order connections, as should be clear from the program trace in the last section. This will be shown again in Section 7.3.5.

Training Algorithm	Dimension of Difficulty							Incremental	Constructive	Order > 1	Order > 2	
	1	2	3	4	7	8	9					
Temporal Transition Hierarchies	D	C	LS	M/M	M−k	k+	F	√	√	√	√	

Table 6.4: The characteristics of the Temporal Transition Hierarchy algorithm. The categories are the same as those of Tables 3.1 and 4.2. "LS" in column 3 indicates that, given knowledge of the previous state, desired mappings from senses to actions must be linearly separable. All other notation is identical to that of Table 4.2.

functions sometimes raise a red flag in the connectionist community due to their weak powers of discrimination in traditional networks (Section 3.1). The same holds for networks without hidden layers. However, transition hierarchies use higher-order connections, involving the multiplication of input-units with each other (Equations 6.3 and 6.4). This means that despite the linear activation function and lack of hidden units, the network can in fact compute non-linear functions and can make classifications that are not linearly separable. Nevertheless, these mappings are constructed from the inputs at previous time steps; without previous inputs, the network can only generate linear outputs from its input at the current time step.

This need not be a problem, however. If the network's input is repeated over multiple time steps, the network can compute non-linear mappings from the repeated input data. In robotics environments this can be done by giving the robot a "stay" action, allowing the agent to stay in its current position until it has made whatever discrimination it needs to. This approach is discussed in Section 7.3.5. Different possible solutions are discussed in Section 8.4.

Another issue is that of unlimited time-delays. The system described above is only capable of building hierarchies that span a fixed number of time steps. This means it can only learn Markov-k tasks (Section 2.2.3), though it can learn them when k is unknown. It cannot learn arbitrary finite-state grammars or any more complicated tasks. A possible solution for this limitation is discussed in Section 8.4. This is also not necessarily an enormous drawback, since the algorithm can still learn k regardless of its size.

A final issue is that of reinforcement learning, which was handled so intuitively in the case of behavior hierarchies in Section 6.1.4. Since the units of transition hierarchies are not dedicated to representing entire behavior sequences, the technique used in that section cannot be used here. Fortunately, standard reinforcement learning techniques *may* be used. Since the algorithm of Section 6.2.5 can be used in any supervised-learning task, it can be used to predict critic values or Q-values as described in Chapter 5. Results with this system are presented in Section 7.3.

Dimensions of Difficulty. Table 6.4 summarizes the Temporal Transition Hierarchies learning algorithm in the same form as given in Tables 3.1 and 4.2.

Chapter 7

Simulations

Of the two algorithms presented in Chapter 6, Temporal Transition Hierarchies is clearly superior. It has been tested on a variety of learning tasks, and the results of these tests are reported here.

Experimental results are presented in two categories: supervised-learning benchmarks and continual-learning demonstrations. There are two supervised-learning benchmarks: The Reber grammar and the Mozer *gap* task. In these tasks, the learning system is given a sequence of data for which its task is to predict the next item in the sequence. For both benchmark tasks the next item cannot be predicted from the current item alone but must take into account some amount of previous information as well. The amount of previous history required is not given but must also be learned. One of the supervised tasks, the *gap* task, is deterministic; the other, the Reber grammar is stochastic.

The remaining results demonstrate the ability of temporal transition hierarchies to do continual learning. This is shown through a series of nine small mazes whose states are ambiguously labeled. The mazes are arranged such that each is somewhat more complex than its predecessor. As the sequence progresses, the mazes increase in size, but each preserves the basic structure of its predecessors so that skills learned while solving one maze can be used for solving the next. Therefore, most units created to learn one maze should still be useful when learning the next.

The supervised-learning tasks are benchmarks by which Temporal Transition Hierarchies can be compared with other sequence-learning neural networks. The continual-learning tasks are original and demonstrate the concept of continual learning while simultaneously demonstrating the success of Temporal Transition Hierarchies in non-Markovian environments. Table 7.1 shows how difficult the tasks are according to the eleven dimensions of difficulty from Table 2.1. All are Markov-k tasks. All contain ambiguous sensory information. Only the maze tasks require reinforcement learning. The mazes can all be *solved* with a small amount of state information. The difficulty of *learning* to solve the tasks, however, is great.

7.1 Description of Simulation System

The simulations described in this chapter were done with a highly modular, object-oriented architecture. The architecture is designed to allow any combination of environment and learning agent by providing a common protocol that all environments and agents must adhere to. The system is diagramed in Figure 7.1. The agent module implements Equations 2.2, 2.3, and 2.7. The environment module implements Equations 2.4, 2.5, and 2.6. The centerpiece is the Agent-Environment Interface, which transfers sensory and reinforcement information from the environment to the agent, transfers action information from the agent to the environment, and mediates in all matters of protocol. The user may specify

	Dimension of Complexity	Reber	Gap Task	Mazes
1	Sense/Action Representation	Local	Local	Local
2	Sense/Action Values	Binary	Binary	Binary
3	Sense→Action Mapping	Orthogonal	Orthogonal	Orthogonal
4	Sense→State Mapping	Many-Many	One-Many	One-Many
5	State→Action Mapping	Many-Many	One-Many	Many-Many
6	Next State Function i.e., (state, action)→state	Stochastic	Deterministic	Deterministic
7	Underlying Model	Markov-k	Markov-k	Markov-k
8	History Information Needed	7	1	0–1
9	History Duration	1	2–40	1
10	State/Action→Reinforcement Mapping	N/A	N/A	Many-one
11	Planning Steps for Reinforcement	0	0	8–27

Table 7.1: The three kinds of tasks for which results were obtained are described in terms of the eleven dimensions of difficulty from Table 2.1. For the gap and maze tasks which are really *sets* of different tasks, ranges of values are reported where applicable.

the environment, the agent, parameter settings for each, and parameter settings for the interface.

The environment's parameters are environment dependent. They describe, for example, the environment's size, its initial random seed (if it is stochastic), the file from which to retrieve its initial configuration, at what level of detail its activities should be displayed, etc. The agent's parameters are agent dependent and might include the agent's numeric parameters (e.g., α, β, γ, etc.), its display level, which learning algorithm to use, and the learning algorithm's parameters (e.g. η, Θ, ϵ, display level, etc.). The interface parameters specify control information such as the maximum number of steps per trial, the maximum number of trials, and stopping criteria for training and testing. After these parameters are specified, the interface is called to do the training or testing and can be queried afterwards for the results.

7.2 Supervised-Learning Tasks

The two supervised-learning tasks reveal the strengths of Temporal Transition Hierarchies in two kinds of environments. The Reber grammar [79] demonstrates the reliability of the algorithm when learning temporal dependencies in the face of noise. It has been used often in the neural network literature to demonstrate the relative power of various recurrent-network approaches. The *gap* task introduced by Mozer [66], demonstrates the system's ability to learn long temporal dependencies reliably and quickly.

7.2.1 Reber Grammar

The Reber grammar is a small finite-state grammar with one or two possible transitions from every state (Figure 7.2). Transitions from one node to the next are made by way of the labeled arcs. Starting in state 0, training strings are generated by randomly choosing an outgoing arc from the current state (in states 0 and 7, only one arc may be chosen). The chosen arc is then traversed to the next state. For example, if the grammar is in

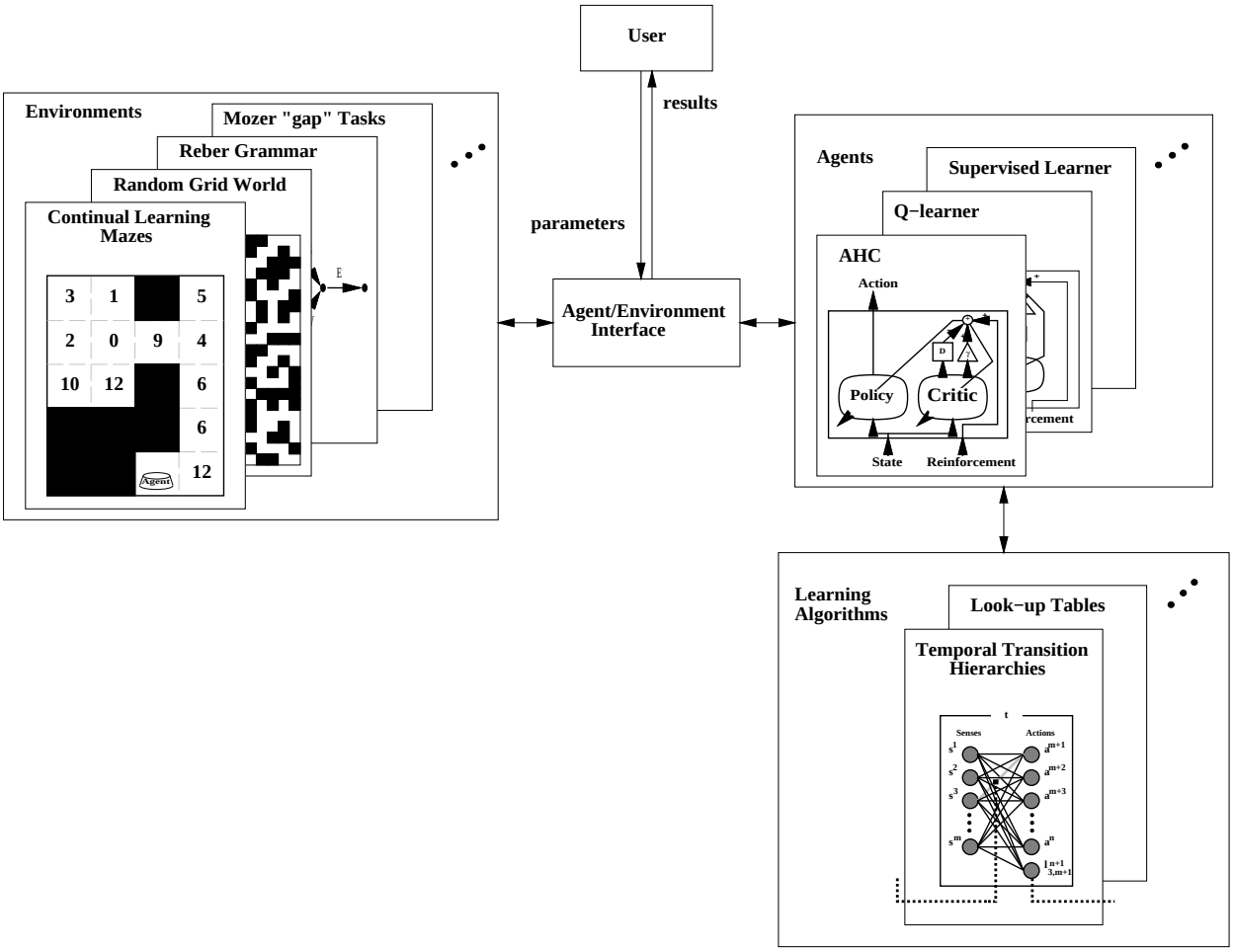

Figure 7.1: The simulator has three primary components: the Agent, the Environment, and the Agent-Environment Interface. The Agent may make use of one or more learning algorithms, possibly chosen by the user. The Agent-Environment interface is given all user-specified options and parameters before training begins, and can be queried afterwards for the results.

state 1, either arc T or P may be chosen. If T is chosen, state 2 is reached, and from there either S or X is chosen. This process continues until state 7 is reached. The sequence of arcs traversed makes up a training string. "BTSXXTVPSE" is a sample string generated by the grammar.

The task of the network is to predict the next element in the string while the string is presented, one element per time step, to the network as input. Both inputs and outputs are encoded locally; so there is exactly one input unit and one output unit for every possible arc label. This means there are seven input and output units (one each for B, T, S, X, V, P, and E). One input unit is set to 1.0 at every time step. With the above string, the first input would be a B (the B input unit would be set to 1.0, all others to 0.0), and the target would be T (the target value for the T output unit would be 1.0, all others would be 0.0). The next input is T, and the target is S. The third input is S, and the target is X, etc. The Reber grammar is difficult for two reasons: (1) because the current state cannot

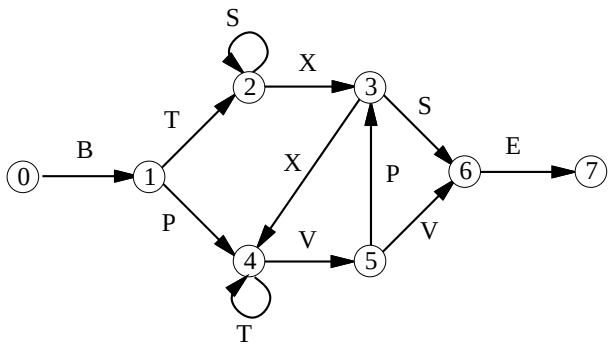

Figure 7.2: The "Reber" grammar is the finite-state grammar shown above. Each numbered state has one or two outgoing arcs. The current state cannot be determined from knowledge of just the last arc traversed.

be determined from the current input alone, and (2) because the next target cannot be completely determined from the current state.

The Temporal Transition Hierarchy network was trained on this task using the following method. Before each training string was presented, the network was reset — the values of all units were set to zero and the record of previous activations was cleared. A newly generated, random string was then presented. The network was trained until it had correctly predicted 100 *consecutive strings*. A string was considered to be correctly predicted if the prediction at every time step was correct. However, since in most states there are two possible arcs that can be traversed, the network is considered to have made a correct prediction if its one or two most highly activated units matched the one or two possible arcs that might have been generated next. If the initial B were presented from the string above, then the prediction would have been considered correct if the two most highly activated output units were P and T (even though the target would be T alone). If only one arc can be traversed from a state, it must correspond to the single most highly activated unit. (The actual level of activation is of no consequence; only the relative level of activation among the units was tested.) When 100 consecutive sequences were correctly predicted, training was stopped. These 100 strings may be thought of as a validation set in which training continues. (The number 100 was chosen more or less arbitrarily, but large enough to ensure good performance on the test set). The network was then tested on a set of 128 freshly generated sequences. (This number was chosen for easy comparison with Recurrent Cascade Correlation).

Table 7.2 presents the results of 100 different learning *episodes*, where an episode is the entire training process as just described. Each episode uses different randomly-generated strings (i.e, the random number generator was seeded differently for all 100 episodes). The table shows the number of strings seen by the network during training *including* the 100 correctly classified consecutive validation strings. The mean performance, best performance and standard deviation are shown, together with the number of units created and average percent of testing strings correctly predicted. Also shown are the corresponding values (when available) of other systems, quoted from reports published elsewhere (in most cases by the system's inventors).

An Elman-type recurrent network was able to learn this task after 20,000 string presentations using 15 hidden units [17]. (The correctness criteria for the Elman net were

Algorithm:	Elman Network	RTRL	RCC	Temporal Transition Hierarchies
Training Strings (Best):	20,000	19,000	–	127
Training Strings (Mean):	–	–	25,000	167.7
Standard Deviation:	–	–	–	35.1
Hidden or High-Level Units:	15	2	2-3	40
Percent Testing:	100%	100%	100%	100%

Table 7.2: Temporal Transition Hierarchies are compared against recurrent networks on the Reber grammar. The results for the recurrent networks are quoted from other sources [17, 27]. The mean and/or best performance is shown when available. RTRL is the Real-Time Recurrent Learning algorithm [123]. RCC is the Recurrent Cascade Correlation algorithm [27].

slightly more stringent than those described in the previous paragraph: the output units corresponding to the one or two possible next arcs had to have an activation level of at least 0.3). Recurrent Cascade Correlation (RCC) was able to learn this task using only two or three hidden units in an average of 25,000 string presentations [27]. The transition hierarchy system learned the task in an average of 167.7 strings, but had to be constrained not to create more than forty units. Had no constraint been imposed, the system would have continued to add new units in an effort to better predict the randomly selected arcs. In other words, it would have spent limitless resources trying to predict the random number generator. This is an unavoidable situation, since there is always the possibility that some input from the arbitrarily distant past will finally make predictable a previously random-seeming event.

The parameters used for the results in Table 7.2 were: $\eta = 0.06$, $\eta_L = 0.15$, $\sigma = 0.25$, $\Theta = 0.9$, $\epsilon = 0.0$. Considerable efforts were made to optimize these values. The system was trained repeatedly on a fixed training set using a multidimensional direction set method (Powell's method [77, §10.5]) — an optimization technique for use in the absence of gradient information. The technique can be used to find very good parameter settings without the hassle and distress that tend to accompany a manual search for good values. I recommend this technique to all my friends.[1]

Typically, the optimization routine tests many hundreds, sometimes thousands of parameter combinations. For every parameter combination in the Reber grammar task, the network was trained twenty times. The average number of strings seen before passing the 100-consecutive-strings criteria was then added to the number of higher-level units created. This sum was the value the optimization routine tried to minimize.

The optimization routine could be used because, fortunately, the transition hierarchies

[1]This method is also very computation intensive, and can really only be used with very fast learning algorithms (or very easy learning tasks). Though the direction set method is very useful in optimizing parameters, its usefulness is due mostly to its convenience. It has no magical abilities to find exceptionally good parameters. It simply carries out automatically what one would normally do by hand, and it gives one confidence that the space has been searched adequately. The best improvement I've seen this method find over a coarse manual search is about 20%.

	20%	50%	90%	99%	100%	101%	110%	200%	500%
η	19/ 0	6.3/ 0	-2.3/ 0	0.0/ 0	0.0/ 0	-0.7/ 0	-0.9/ 0	56/ 3	570/ 7
η_L	-2.9/ 1	-5.4/ 0	-1.2/ 0	0.0/ 0	0.0/ 0	0.0/ 0	0.5/ 0	0.6/ 0	7.3/ 0
σ	23/ 0	-2.4/ 0	-3.0/ 0	0.0/ 0	0.0/ 0	0.6/ 0	-0.9/ 0	100/ 3	440/14
Θ	310/11	39/ 3	-4.0/ 0	0.6/ 0	0.0/ 0	0.0/ 0	-1.5/ 0	-3.5/ 0	4.2/ 0
	-0.5	-0.2	-0.1	-0.01	0.0	+0.01	+0.1	+0.2	+0.5
ϵ	260/ 9	36/ 2	34/ 1	0.1/ 0	0.0/ 0	-0.2/ 0	-0.4/ 0	20/ 1	740/25

Table 7.3: The impact of changes in the parameters while training on the Reber grammar. The middle column, labeled 100% shows the result of training on the parameters found through optimization. The other columns each display the effect of modifying the parameter named in the left column by the amount shown in the column heading. The values listed show the percent effect on training performance. Because ϵ had an optimized value of 0.0, it is changed to the shown value rather than being modified by a certain percentage. The second value in each column is the number of training episodes that did not achieve perfect testing.

algorithm is very fast. On a Sparc 10, the average amount of time spent for one complete episode of training and testing on the Reber grammar (as described above) was 0.70 seconds (when the average number of strings used for training was 167, and the number of testing strings was always 128). However, doing this twenty times for each parameter setting still takes quite a while. When optimization finally stopped, the resulting parameters were used to train and test the system 100 times on newly generated data. These are the results that appear in the table.

Parameter Sensitivity. One drawback of automatic parameter search is that the sensitivity of the algorithm to changes in parameters becomes somewhat hidden. In order to give at least some feeling for this algorithm's parameter sensitivity, it was trained on the Reber-grammar task with a barrage of different settings. These are shown in Table 7.3. Only one parameter is changed at a time; all others are kept at their optimized values. The table shows the effect of modifying each parameter. For each setting, the network was trained on the Reber grammar 25 times and the performance was averaged. Performance was measured in terms of the number of strings seen by the network up until 100 consecutive strings had been correctly classified. The value displayed is the percentage difference from the performance of the network with all parameters optimized. Also displayed is the number of training episodes (out of 25) that did not result in perfect testing. The testing set, as before, consisted of 128 randomly generated strings.

The sensitivities are varied, but in general the algorithm is quite robust with respect to its parameters. For example, reducing η by a factor of five (to 20% of its optimized value) results in only a 19% slow down. (Though increasing it by a factor of five results in a slow down of 570%.) With less extreme changes, however, the slow downs are very reasonable. Even a factor of two change in either direction did not result in more than a factor of two slow down for any of the parameters. With changes of as much as 10%, the differences are negligible. (The reason that some parameter settings performed even better than the optimized values is because they were not tested on the same training data, and differences in training data can have a noticeable effect on performance).

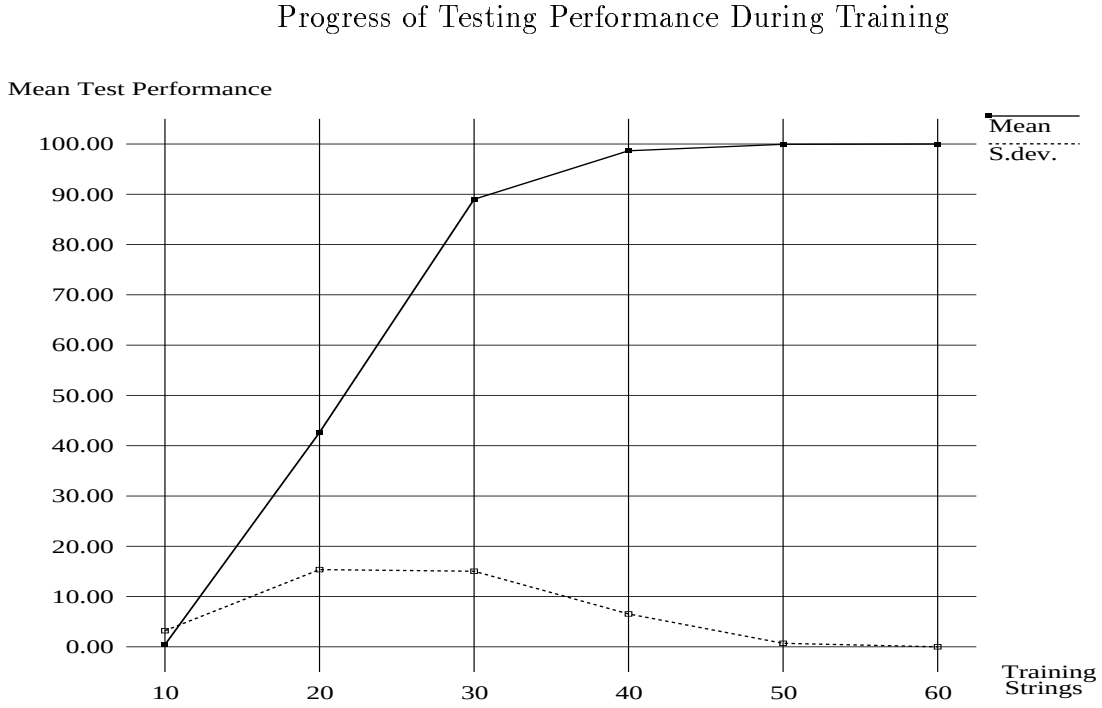

Figure 7.3: The horizontal axis shows the number of training strings seen by the network. The vertical axis shows (1) the mean percent correct, and (2) the standard deviation on randomly generated test sets over 100 networks.

Performance Improvement During Training. The progressive performance of the network can be seen by testing it after every 10 training strings. A graph displaying this information is shown in Figure 7.3. The graph shows the mean performance of 100 training episodes. Performance was measured as the percentage of the testing set correctly predicted. The test set consisted of 128 randomly generated strings. On average, the network can predict 43% of the testing set correctly after 20 training strings. After 40 training strings, the average performance is 99% correct. After 50 training strings this value is 99.9%, and the network achieves 100% after 60 training strings. The standard deviations are shown in the lower curve. (The network seems to perform better in Figure 7.3 than in Table 7.2 because the validation strings are included in the latter's values.)

As stated previously, the network's error tends to decrease fairly monotonically. This can be seen (with eyes squinted) in Figure 7.4. Training is not epoch-wise, but is done on-line as strings are presented, so the RMS error is shown for each string. Because of this and because of the stochasticity in the environment, (i.e., the strings are of many different lengths) this is about as close to a monotonic decrease as one could expect. In a non-stochastic environment with a fixed training set, a less chaotic graph emerges, as is shown in the next section.

7.2.2 The *Gap* Task

Temporal Transition Hierarchies have also been tested on the *gap* tasks introduced by Mozer [66]. These tasks test the ability of a learning algorithm to bridge long time delays.

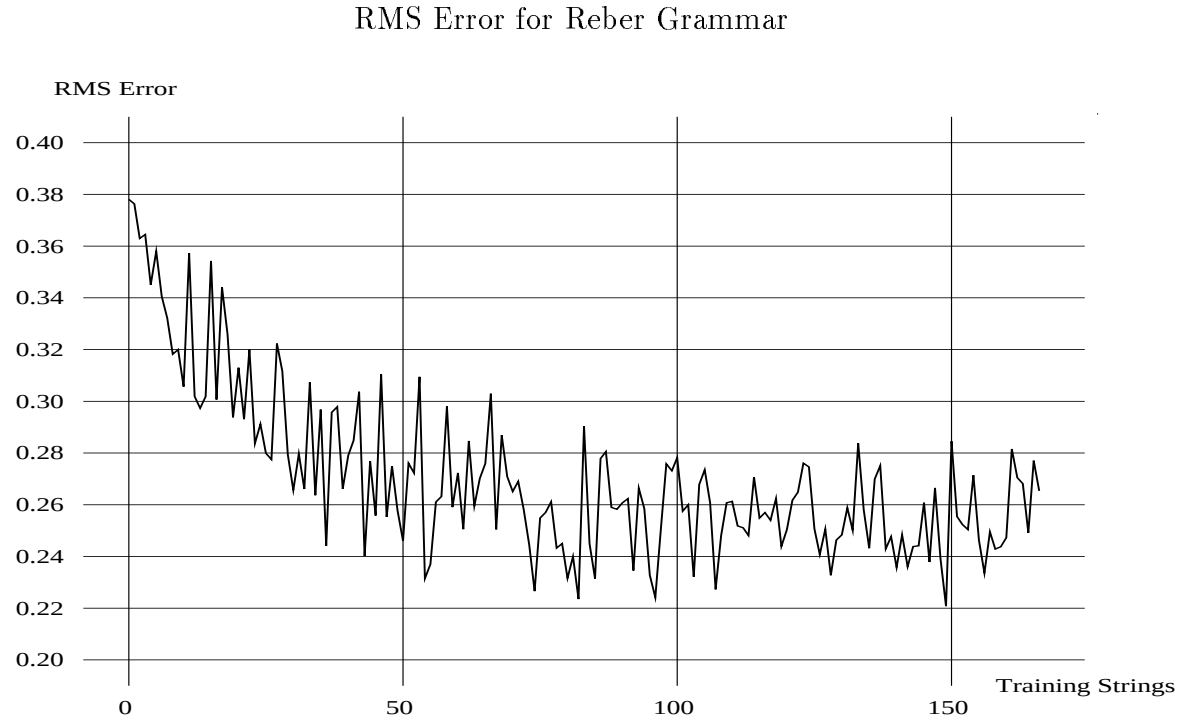

Figure 7.4: A typical graph of RMS error while learning the Reber grammar. In this training episode, the algorithm saw 166 strings. It predicted the last 100 strings correctly.

Each task consists of a two-string training set such as that shown in Figure 7.5. Each string consists of a series of forty-three input elements. As with the Reber-grammar strings, the network must learn to predict the next element of the sequence. Each string is presented to the network one input at a time (with the following input as the target) until the network can predict both sequences correctly. A sequence is predicted correctly if the highest activated output unit corresponds to the next item in the sequence for every element after the first.

The gap task is difficult because the two sequences are nearly identical. Only one item distinguishes them: the initial input element (an X or Y), which is presented again after a gap of several other inputs. In order to predict its second appearance, the initial element must be remembered across this gap while the other items are presented. The task can be made more difficult by increasing the size of the gap. In Figure 7.5 the gap is two, in Figure 7.6 the gap is 8. Other than the placement of the X's and Y's, the strings are identical

time step:	0	1	2	3	4	5	6	7	8	9	10	...
Sequence 1:	X	a	b	X	c	d	e	f	g	h	i	...
Sequence 2:	Y	a	b	Y	c	d	e	f	g	h	i	...

Figure 7.5: An example of a *gap* training set [66]. One item is presented to the network at each time step. The target is the next item in the sequence. Here the gap is two because there are two items in the sequence between the first X or Y and the second X or Y. In order to correctly predict the second X or Y, the network must remember how the sequence began.

time step:	0	1	2	3	4	5	6	7	8	9	10	...
Sequence 1:	X	a	b	c	d	e	f	g	h	X	i	...
Sequence 2:	Y	a	b	c	d	e	f	g	h	Y	i	...

Figure 7.6: An example of the *gap* task with a gap of 8.

in all tasks. The inputs are locally encoded (one input unit for each unique sequence item).

Results of the gap tasks are given in Table 7.4. The values for the standard recurrent network and for Mozer's own variation are quoted from Mozer's paper [66]. Mozer, whose network was specifically designed to learn long temporal dependencies (Section 4.3.3), reported results for gaps up to ten, and he also stated that the network had scaled linearly even up to gaps of 24. His network required different parameter settings for each gap size and found that the parameters needed very precise tuning for the largest gaps. The parameter settings for transition hierarchies, on the other hand, were constant across all gap sizes. For the results in Table 7.4, they were: $\eta = 0.8, \eta_L = 0.01, \sigma = 0.5, \Theta = 1.0$, and $\epsilon = 0.1$. No bias unit was used.

A graph of the RMS error for gaps of two, ten, twenty, thirty, and forty is displayed in Figure 7.7. Unlike with the Reber grammar, the gap tasks are deterministic, and a fixed training set is used. This allowed the fast, monotonically non-increasing RMS error. All gap sizes plateau at an RMS minimum until the last needed units are finally built. At this point the error suddenly dives again.

With a gap of two, the following high-level units are created after seeing both patterns once:[2] $l^1_{a,X}, l^2_{X,b}$ and $l^3_{a,Y}$. The first and third of these helps the network to predict whether a or c will appear after X or Y is seen. The second helps decide whether X or Y should follow b. After the next pass through the training set, four new units are created: $l^4_{c,X}, l^5_{Y,b}, l^6_{c,Y}$ and $l^7_{2,a}$. units 4 and 6 will work in cooperation with units 1 and 3: units 1 and 4 will allow the network to predict whether a or c should follow X; units 3 and 6 do the same for Y. Unit 5 does for Y what unit 2 did for X. Unit 7 is now the next stage in the chain that will reach back to the initial X and Y. It will determine whether unit 2 should be activated after a is seen. If activated, unit 2 will predict X if b is seen. In the next pass, only one new unit is created: $l^8_{5,a}$, which does for Y what unit 7 does for X. Now all needed units are in place, and only the weights need to be trained, which they do in two more passes. The final effect is that there is a strong connection from X to unit 7, which causes unit 2 to be activated, which then predicts X following b. Likewise, a strong connection develops from Y to unit 8, which causes unit 5 to be activated, which then predicts Y following b. Meanwhile, units 4 and 6 get a strong weight from b, while units 1 and 3 develop negative weights from b, so that c is predicted following the second appearance of X or Y (and a is predicted following their initial appearance).

The above was with a gap of 2. With a gap of 4, nearly exactly the same thing happens, only the names are changed. d and e, the letters immediately preceding and following the second occurrence of X and Y, replace b and c in the above paragraph. (This takes care of units l^1–l^6.) Higher-level units 7 and 8 are slightly more complicated, but, just as in the last paragraph, they are the first link in the chain back toward the beginning of the sequence.

[2]To reduce confusion, I will label these l units beginning with the index 1, though in practice, the first l unit will actually have an index equal to that of the final output unit plus one

Gap	Mean Number of Training Epochs			Units Created by Temporal Transition Hierarchies
	Standard Recurrent Net	Mozer Net	Temporal Transition Hierarchies	
2	468	328	4	8
4	7406	584	6	12
6	9830	992	8	16
8	> 10000	1312	10	20
10	> 10000	1630	12	24
...			...	...
24	–	–	26	52
...			...	...
40	–	–	42	84

Table 7.4: The Temporal Transition Hierarchy network is compared on the "variable gap" task against a standard recurrent network [88] and a network devised specifically for learning long time delays [66]. The comparison values are quoted from Mozer [66], who reported results for gaps up to ten.

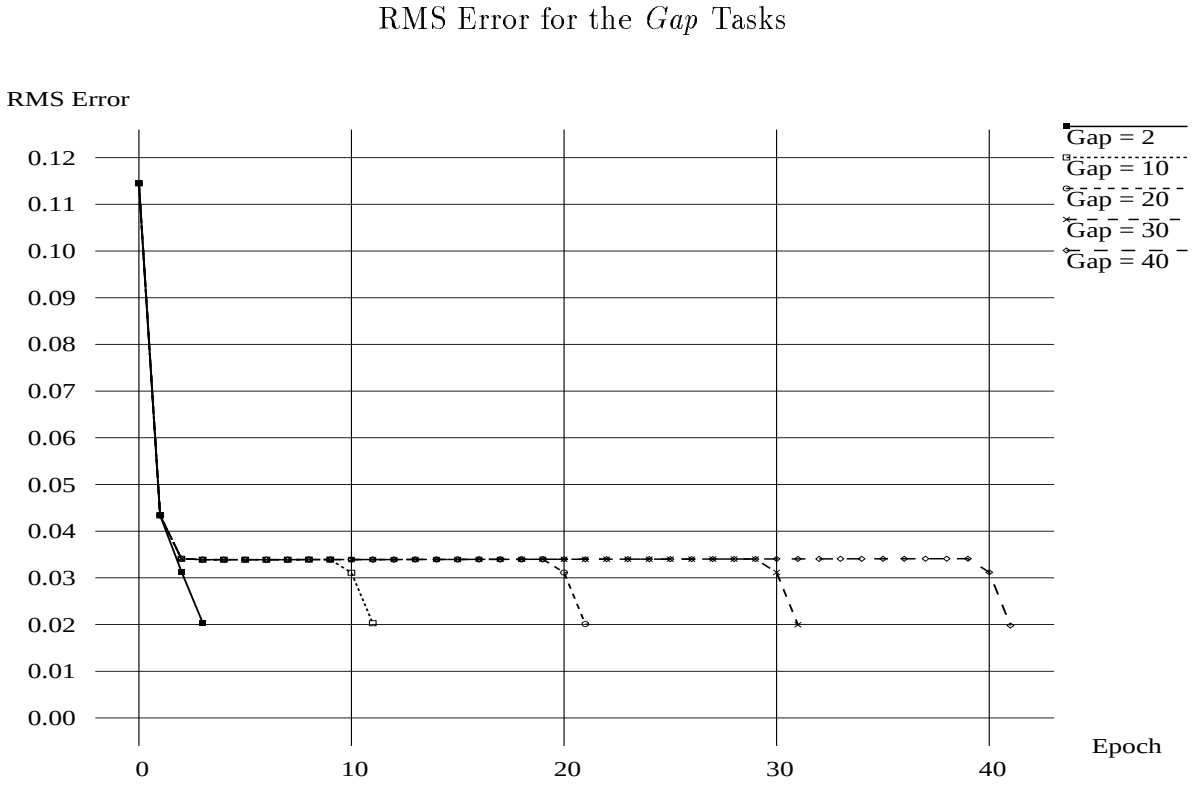

Figure 7.7: The RMS error while learning five different gap tasks is monotonically non-increasing. Gaps of two, ten, twenty, thirty, and forty are shown. Note the highly uniform behavior among all gap sizes. Training stopped once the entire sequence (except for the first element) was correctly predicted.

Thus, with a gap of four, they are $l^7_{2,c}$ and $l^8_{5,c}$. Still more units are needed to reach back to the beginning. Two more are created: $l^9_{7,b}$ and $l^{10}_{8,b}$, which extend the chain back to b. After another pass, the final two units are in place: $l^{11}_{9,a}$ and $l^{12}_{10,a}$. After two more passes, the weights are correct: X activates unit 11, and Y activates unit 12. Then, when at the beginning of the sequence, the a at time step 1 activates either unit 9 or unit 10, so that in the next step, either unit 7 or 8 gets activated, then 2 or 5 in the following time step, and finally at the end of the chain, output unit X or Y gets activated and accurately predicted.

With larger gaps these chains must grow longer. The bridge grows by two units with every pass through the training set, thus extending both chains (one bridging the gap from X to X, the other from Y to Y) one step backwards with each pass. With a gap of n, this requires n passes through the training set. Add to this the last two passes needed to adjust the weights, and the results in Table 7.4 should now be completely explained.

The temporal transition network scaled linearly with every gap size — both in terms of units and epochs required for training — for all sizes tested up to a gap of forty. Because these tasks are not stochastic, the network always stopped building units as soon as it had created those needed to solve the task.

Why does the algorithm learn so quickly? The reason seems to be the specificity with which problems are addressed. The weights of the network are constantly trying to establish reliable temporal relationships between events. Each connection is attempting to answer the question: to what extent does event 2 follow event 1? (Where event 1 is represented by the connection's input-unit, and event 2 is predicted by the connection's output unit). If a prediction cannot be made, this is manifested through the inability of a weight to determine this relationship. Fortunately, these situations can be detected quickly and easily, and a new unit can be built to help establish the circumstances under which event 2 does follow event 1. The new unit is dedicated to the single narrow purpose of trying to discover information — from the previous time step only — that will answer this question. If the needed information does exist at the previous time step, then learning is very fast, since it is single-layer learning (i.e., there are no hidden units between the inputs and the new high-level units). In contrast, the hidden units of traditional neural networks must perform a complex balancing act: they have no single, specific, easily identifiable purpose. Rather, they must cooperate with many other units, searching together in a constraint-satisfaction process through a vast space for a set of weights that, when taken together, will allow the network as a whole to solve its task. Even RCC, which also trains hidden units individually instead of in a cooperative constraint-satisfaction process, does not build them to solve such specific problems as those for which transition-hierarchy units are built.

7.3 Continual-Learning Results

The objective of this section is to illustrate the strengths and practicality of continual learning in reinforcement environments. This is done by introducing CHILD, an agent capable of *continual, hierarchical, incremental learning* and *development*. CHILD combines an incremental, hierarchical learning algorithm (Temporal Transition Hierarchies) with a reinforcement-learning method (Q-learning).[3] Temporal Transition Hierarchies serve as the

[3]In early comparisons between Q-learning and the AHC, Q-learning was generally superior when used with Temporal Transition Hierarchies.

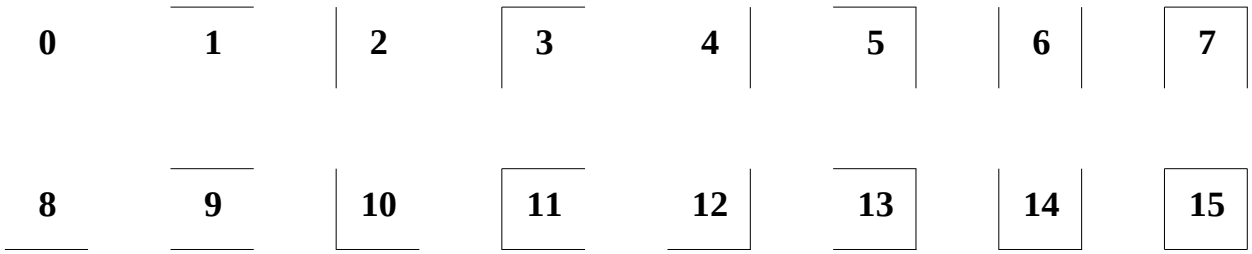

Figure 7.8: There is a unique label for every possible configuration of walls immediately surrounding the agent. "0" means that there are no walls; "1" means there is a wall immediately north of the agent, "2" means there is a wall immediately west of the agent, etc.

supervised learning algorithm with which a Q-learning reinforcement system is constructed. Its capacity to learn in environments with ambiguous sensory information as well as its capacity to do continual learning in those environments is demonstrated in nine successively more difficult maze environments.

In the "maze" environments that follow, there are always four possible actions: move north, move east, move west, and move south. In each state there are senses corresponding to the walls immediately surrounding that state. Thus, there are sixteen possible wall configurations and therefore sixteen unique senses, as shown in Figure 7.8. (Nearly the same sensory system was used by McCallum [59].) Since "15" only occurs if the agent is completely boxed in, it is of little use and does not appear in any of the environments below. Because the Q-learning system has four actions and fifteen senses, the transition hierarchy network therefore must have four output units and fifteen input units.

Learning works as follows. The agent "begins" a maze under three possible conditions: (1) it is the agent's first time through the maze; (2) the agent has just reached the goal in the previous trial; or (3) the agent has just timed out (i.e., the agent took the maximum number of actions allowed for a trial without reaching the goal).

Whenever the agent begins a maze, the learning algorithm is first reset, clearing its short-term memory. In the case of Temporal Transition Hierarchies, this means resetting all unit activations and erasing the record of previous network inputs. A random state in the maze is then chosen and the agent begins from there. Upon arriving in a state, the agent's sensory input from that state is given to the network as input. The network propagates forward to produce a Q-value for each action. The Q-learning system uses these and the current temperature to form a Gibbs distribution (see Section 5.2, Equation 5.7) with which the next action is chosen. The temperature is initially set to 1.0, but its value is decreased at the beginning of each trial to be $1/(1 + n\Delta T)$, where n is the number of trials so far and ΔT is a user specified parameter. The network is updated using the same ΔQ values as given by Lin [54] (Equation 5.8).

The nine mazes are shown in Figure 7.9. Barriers are in black and may not be entered by the agent. Nor may the agent move beyond the boundaries of the maze. The agent receives a reinforcement of 1.0 when it reaches the goal (denoted by the food dish) and receives a reinforcement of 0.0 otherwise. Every state is labeled with the sensory input the agent receives in that state, determined according to Figure 7.8.

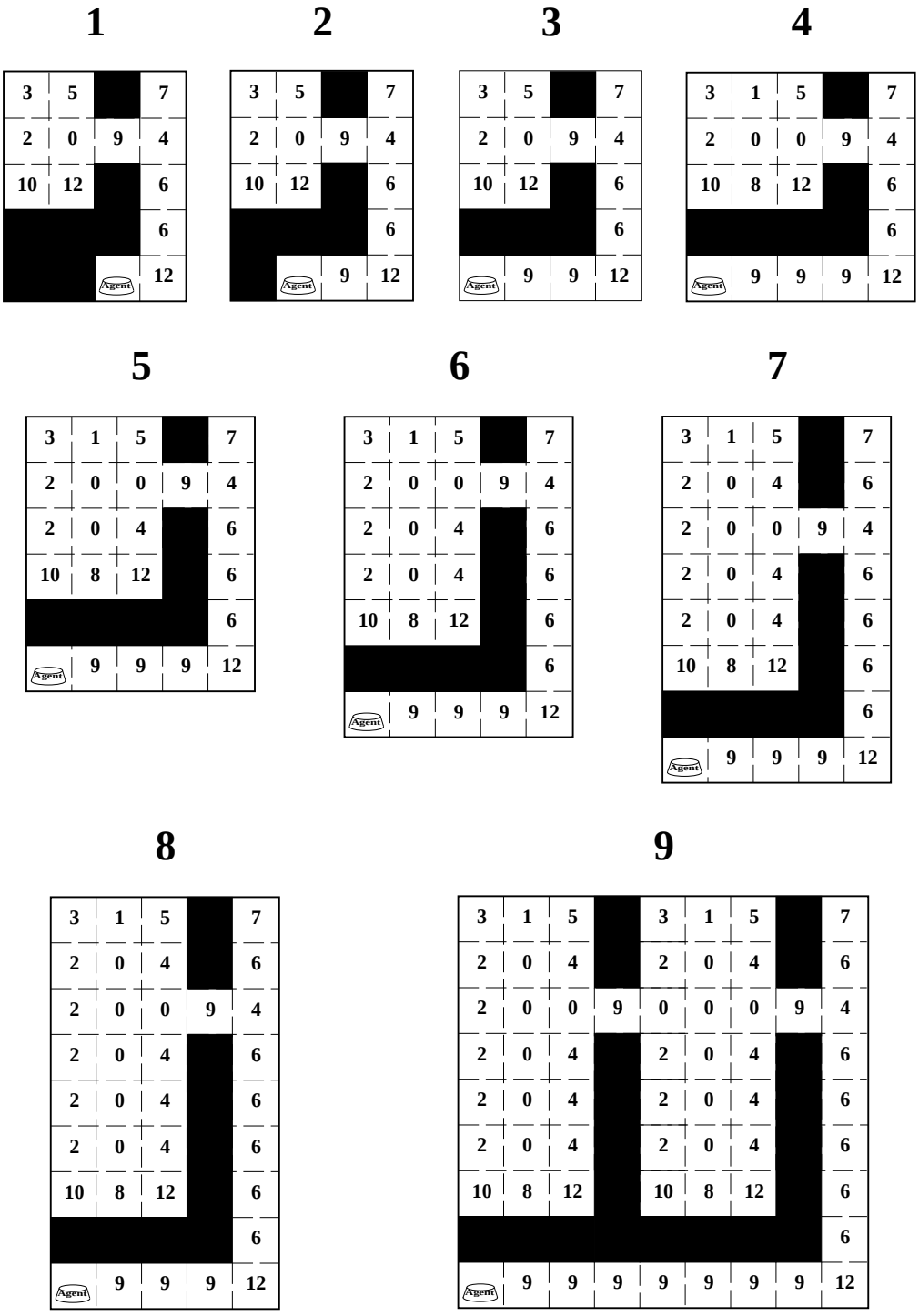

Figure 7.9: These nine mazes form a progression of reinforcement-learning environments from simple to more complicated Each maze is similar to but larger than the last, and each introduces new state ambiguities — more states that share the same sensory input. The digits in the maze represent the sensory inputs as described in Figure 7.8. The goal (a reward of 1.0) is denoted by the food dish in the lower left corner.

Each maze is more difficult than its predecessor, having both more states and more state ambiguities. Though a perfect agent could solve any of these mazes with a very small amount of state information, actually learning the mazes, particularly the last, is quite difficult. Consider, for example, the fact that when the agent attempts to move into a barrier, its position does not change, and it again receives the same sensory input. It is not in any way informed that its position is unchanged. Yet it learns to stop running into barriers nevertheless. On the other hand, in the bottom row of Maze 9 the agent must continue to move east, though it repeatedly perceives the same input. In general, what makes this task difficult is that it is Markov-k only *after* it has been learned. Once the agent's actions are perfect, it can be sure that its previous k senses are sufficient for determining what action to take. Before then, the agent may follow paths that do not preserve needed information (e.g., by moving in small circles). On top of this, the agent has no prior knowledge of the task, including the topology of the mazes, the number of states, what senses each state may have, or what the effects of its actions might be.

7.3.1 Continual Learning vs. Learning From Scratch

CHILD was tested on the nine environments above in two ways: (1) learning each maze independently from scratch, and (2) using continual-learning. In the first case, different agents learn the different mazes independently. In the second, each agent begins in the first and is trained on the others in sequence.

Learning from Scratch. This first set of results demonstrates CHILD's ability to learn in reinforcement environments with ambiguous sensory information. For each environment, multiple agents were created, trained, and tested, and their performance was averaged.

Each agent was given 500 trials to learn a maze and was allowed up to 1000 steps for each trial. After training, the agent was tested for 100 trials. Testing was done *without* stochastic action selection (i.e., the most highly activated action-unit was always chosen), which brings out any obvious flaws in the agent's policy. If, for example, the agent learned to move in a circle, this would not be apparent if it could occasionally take non-policy actions. If its actions are deterministic, however, such a flaw will keep the agent from ever reaching the goal. The average number of steps to the goal was computed for the training phase — timeouts count as 1000 steps. The same was done for the testing phase. (The former measures how quickly the agent learned the task; the latter measures how well.) The former plus the latter, plus the number of units created during training constitute a measure of performance, P. Ten agents were created and tested on each maze with different random seeds, thus creating ten performance measures, $P_0 - P_9$. The average of these was used to indicate the quality of a set of parameters. This was the value that Powell's method tried to minimize. All seven modifiable parameters were optimized for these results. They are given in Appendix C.

As with the supervised tasks, optimized parameter settings for these agents were determined through extensive testing using Powell's method (cf. p. 76). With the optimized parameters for each maze, 100 agents were individually created, trained, and tested in every maze, all with different random seeds. Training was done as follows: each agent was trained for 100 trials and then tested. The agent was considered to have learned the maze if it reached the goal on every testing trial. If the agent did not learn the maze, it was trained

Maze	Mean number of:			Failures
	Training Steps	Units Created	Testing Steps/Trial	
1	1,620	2.61	5.15	0
2	2,362	3.51	5.93	0
3	3,717	3.92	6.66	0
4	6,283	5.58	8.02	0
5	8,856	6.28	9.22	0
6	13,880	7.97	10.75	0
7	13,529	13.93	13.22	0
8	17,833	14.42	14.59	0
9	41,031	21.04	20.73	5

Table 7.5: Results of training and testing 100 agents in each of the nine maze environments shown in Figure 7.9. The first column shows the average number of training steps taken before testing succeeded. The second column gives the average number of units created during training. The third column reports the mean number of steps per testing trial for those cases where training succeeded. The fourth column lists the number of agents that failed to test successfully after 1000 training trials. Only agents that tested successfully were used to calculate the averages in the first three columns.

for 100 more trials and tested again. This process continued until testing succeeded, or until the agent was trained for a total of 1000 trials. The total number of training steps was averaged over all 100 agents in each maze and is reported in the first column of Table 7.5. The average number of l units created is given in the second column, and the average number of testing steps for the last 100 testing trials is given in the third column. There were five failures — five cases where 1000 training trials did not achieve successful testing. All failures occurred while learning the ninth maze. When a failure occurred, the values for training steps, testing steps, and number of units were not averaged in.

The Continual-Learning Case. Rather than learning each maze from scratch, a continually learning agent accumulates what it has learned in each maze to help learn the next. Once an agent has been trained in a maze, it is transferred to the next and training resumes. This is a potentially difficult learning problem. The first maze is relatively easy, since there are only two ambiguous states: those labeled "12." After learning the first maze, the second maze is more difficult for two reasons: (1) there are more ambiguous states (labeled "9"), and (2) many states have changed their distances from the reward, thus requiring slightly different Q-values. These two properties continue to hold for all mazes as each progresses to the next. However, each maze preserves the basic structure of its predecessors so that skills learned while solving one are still potentially useful for the next.

To optimize parameters for this task, ideally several agents would be trained on the first maze, transferred to the second, trained again, etc., until the last maze was learned, and then an average score for these could be minimized with Powell's method. However, the extreme amount of computation required for this procedure effectively precluded optimization. The parameters used are therefore not optimal nor likely even close to optimal, but the results are

Maze	Mean number of:				Failures
	Training Steps	Cumulative Steps	Units Created	Testing Steps/Trial	
1	2,300	2,300	1.46	5.19	0
2	936	3,236	5.13	6.19	0
3	903	4,139	7.46	7.03	0
4	1,306	5,446	10.77	8.21	0
5	3,145	8,592	15.59	9.59	0
6	489	9,081	16.12	10.98	0
7	3,901	12,983	20.72	12.24	0
8	103	13,086	20.83	13.32	0
9	1,139	14,225	21.73	18.85	2

Table 7.6: Results of continual learning in the environments shown in Figure 7.9. After learning one maze, the agent was put into the next and tested. If testing failed the agent was trained on the new maze and tested every 100 trials. If 1000 trials were reached without successful testing, training was stopped, the training was counted as a failure (fifth column), and the number of steps taken was not averaged into the totals in the other columns. A single, non-optimized set of parameters was used throughout training. The first, third, and fourth columns correspond to the first, second, and third of Table 7.5. The second column shows the cumulative number of training steps.

favorable nevertheless and indicate the clear superiority of the continual-learning method.

To compare continual learning to learning from scratch, Table 7.6 presents continual-learning results just as Table 7.5 presented learning-from-scratch results. Exactly the same method was used to produce these, except for two differences: (1) after learning one maze, the agent was transferred to the next maze in the series, and (2) an agent was tested in the new maze *before* training; if the agent already met the testing criteria, it was not trained but went immediately to the next maze. T was reset to 1.0 when training began in a new maze. This regime was applied 100 times and the results were averaged (as in Table 7.5). There were two failures in the continual-learning case. Both occurred in the ninth maze, where the agent did not learn the maze despite 1000 training trials.

Three graphs comparing the performance of continual learning and learning from scratch are given in Figure 7.10. In all cases there are two lines, one for continual-learning and one for learning from scratch. In Figure 7.10A, there is a third line showing the *cumulative* number of steps taken by the continual-learning agent since the beginning of training, i.e., the values from the second column of Table 7.6.

The parameters for all mazes were: $\eta = \beta = 0.3$, $\eta_L = 0.09$, $\sigma = 0.3$, $\Theta = 0.56$, $\epsilon = 0.11$, $\gamma = 0.91$, $\Delta T = 2.1$.

Analysis. The graphs clearly demonstrate the advantages of continual learning. The learning-from-scratch agents outperformed the continual-learning agents on the first maze due to the optimized parameters. After the first maze, however, the continual-learning agents always learned faster. In fact, after the third maze, despite the disparity in parameter optimality, even the *cumulative* number of steps taken by the continual-learning

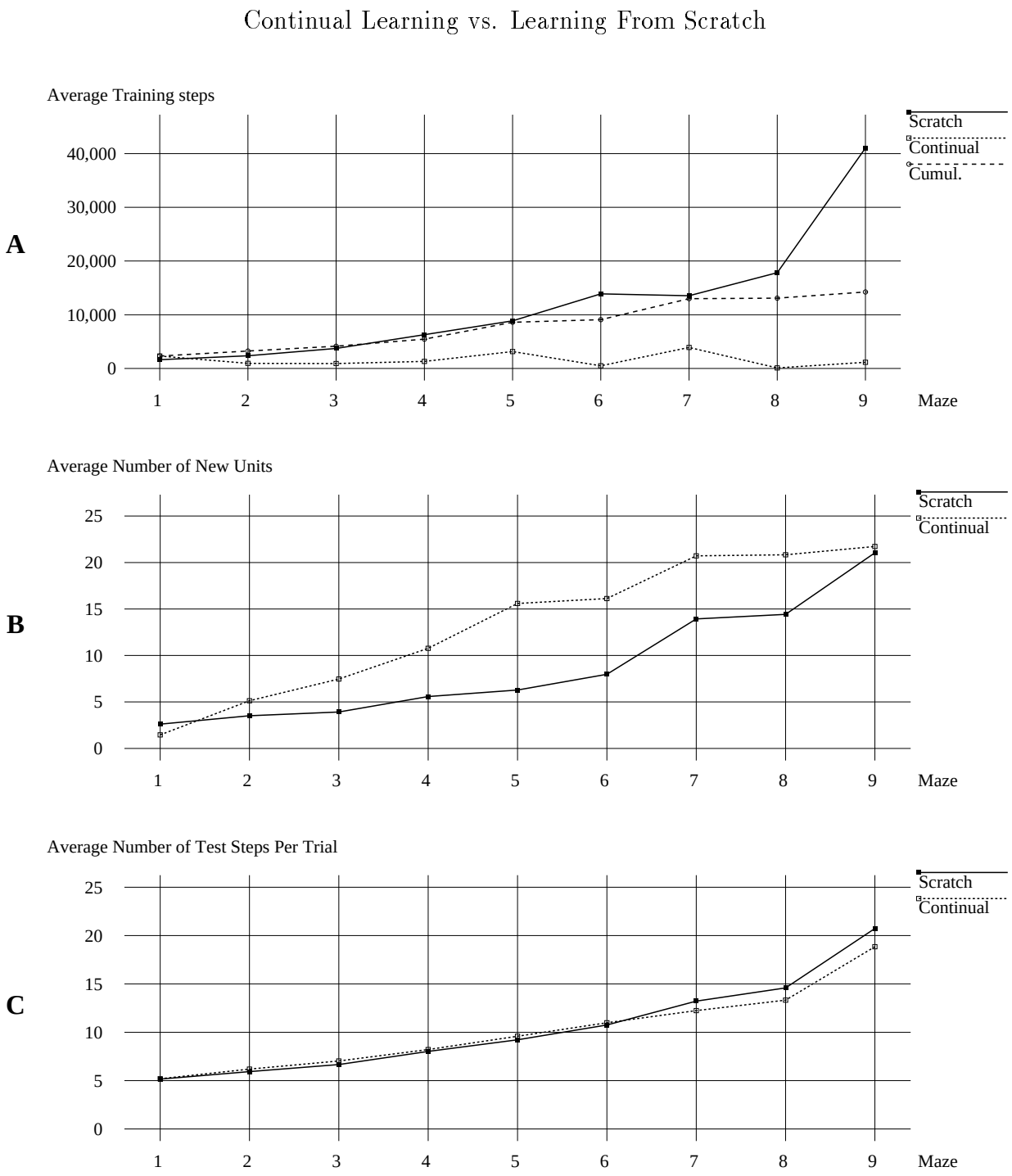

Figure 7.10: Graph (A) compares Learning from Scratch with Continual Learning on the nine Maze tasks. The middle line shows the average cumulative number of steps used by the continual-learning algorithm in its progression from the first to the ninth maze. Graph (B) compares the number of units created. The line for the continual-learning case is, of course, cumulative. Graph (C) compares the testing performance of both methods. The values shown do not include cases where the agent failed to learn the maze.

agent was less than the number taken by the agent learning from scratch. By the ninth maze the difference in training became drastic. The number of extra steps needed by the continual-learning agents was tiny in comparison to the number needed without continual learning. The cumulative total was about a third of that needed by the agents learning from scratch. Furthermore, the trends shown in the graphs indicate that as the mazes get larger, as the size and amount of ambiguity increases, the difference between continual learning and learning from scratch increases drastically.

It would not be too surprising if the cumulative training figures for the continual-learning agents were comparable with the training figures of the learning-from-scratch agents, (and it would also not be too surprising if the earlier learning had interfered and caused the continual-learning agents to be slower). But the earlier training significantly *enhanced* the performance of the later learning. This demonstrates not only that, given a sufficiently sophisticated agent, continual learning can dramatically reduce the training required for complicated tasks, but also that CHILD is able to take advantage of such training procedures and is truly capable of continual learning in reinforcement environments.

Besides training, testing also compares favorably for the continual learner: after the sixth maze, the continual-learning agent found better solutions as well as finding them faster. This is perhaps attributable to the fact that, after the first maze, the continual-learning agent is always in the process of making minor corrections to its existing Q-values, and hence to its policy. The corrections it makes are due both to the new environment and to errors still present in its Q-value estimates. The number of units needed seems to escalate at first for the continual-learning case, but then levels off after a while, showing that the units created in the first eight tasks were mostly sufficient for learning the ninth. The fact that the learning-from-scratch agent created about the same number shows that these units were also necessary.

It is extremely common for comparisons to be done between two methods where one is the favorite and the other is a straw man waiting to be shown up by the favorite. Achieving good results in this situation is usually not difficult, but also rarely reliable or scientific. Even if one tries to be fair, the straw man is always too easy to blow over. I've tried to avoid that by using an objective source to make the contending approach (in this case, learning from scratch) as strong as possible. The optimization method is unbiased in its pursuit of good parameters, and in my experience it has always done somewhat better than the best I could do by hand.

7.3.2 Proprioception

In order for CHILD to learn a policy for an ambiguous environment, say Maze 8, it must use previous sensory information, since current sensory data is insufficient. Take for example position (3, 5) in Maze 8 (three places from the left, five from the bottom). The sensory input is the same as that in the squares immediately south and immediately north. If the agent's current input is 4, how can it predict what it will see if it takes a step north or south? Even more difficult: if its previous two inputs were *also* both 4, what can it predict? If its previous *actions* are not also known, it has no way of discriminating (3, 4) from (3, 5) and (3, 6). The agent can in general only make these predictions if its actions are *consistent*; i.e., if having seen two successive 4's the agent always moves north, then upon seeing a third, it can assume its last move was a move north.

Maze	Mean number of:			Failures
	Training steps	Units Created	Testing steps/Trial	
1	2,984	5.92	6.21	0
2	5,707	9.84	7.26	0
3	4,440	4.74	7.59	0
4	5,111	6.78	9.11	0
5	8,146	5.72	11.25	1
6	14,537	8.92	13.70	0
7	9,980	10.02	14.64	3
8	14,350	12.75	16.46	0
9	38,153	14.51	24.82	1

Table 7.7: Results of using proprioception when training Mazes 1–9 from scratch. The meanings of the columns correspond to those in Table 7.5.

To help CHILD better predict its Q-values, its actuators were made proprioceptive; i.e., its last action as well as its current environmental stimulus were given to the agent as input. This way, its predictions are based upon a better representation of its previous experiences and can therefore be more accurate. With proprioception, the inputs were still encoded locally; with 15 senses and four actions there were 60 network inputs, only one of which was active at a time. As with the non-proprioceptive case, optimization was done for Mazes 1–9 individually but not for the continual-learning case.

Tables 7.7 and 7.8, like Tables 7.5 and 7.6, compare the performance of the learning-from-scratch agent and the continual-learning agent. The parameters are given in Appendix C.

Figure 7.11 compares the continual-learning and learning-from-scratch data graphically, just as was done in Figure 7.10. Here, the continual-learning case failed once in Maze 1 (which consequently made learning impossible in all later mazes); the learning-from-scratch case failed once in Maze 5, three times in Maze 7, and once in Maze 9.

Maze	Mean number of:				Failures
	Training steps	Cumulative Steps	Units Created	Testing steps/Trial	
1	4,808	4,808	6.48	7.19	1
2	635	5,443	10.14	8.11	0
3	36	5,479	10.18	8.62	0
4	1,362	6,842	15.76	10.00	0
5	3,010	9,853	21.27	11.48	0
6	754	10,607	22.16	13.21	0
7	1,388	11,996	23.86	14.03	0
8	0	11,996	23.86	15.60	0
9	126	12,122	24.02	22.09	0

Table 7.8: Results of continual learning in Mazes 1–9 when using proprioception. The meanings of the columns are the same as those in Table 7.6.

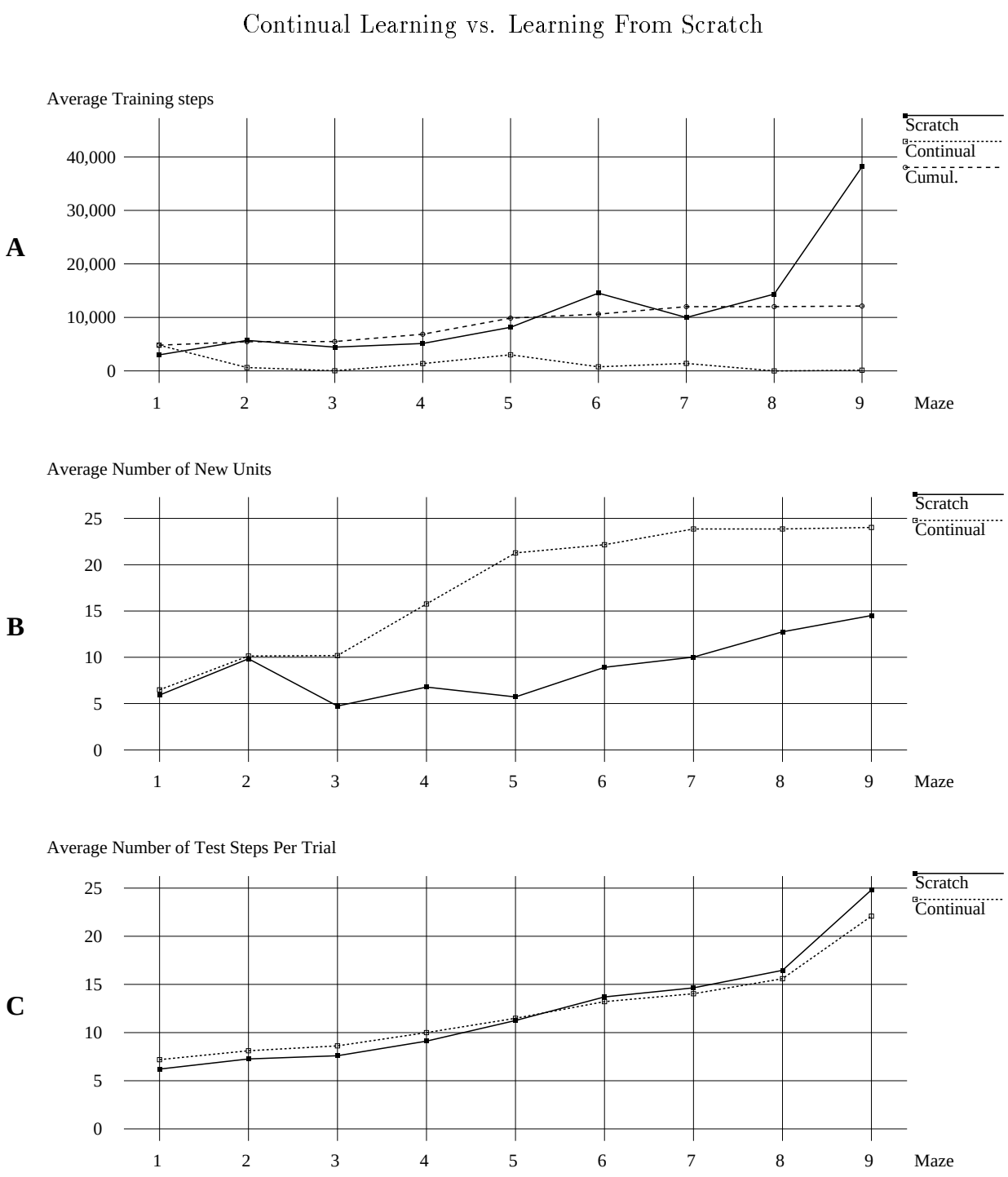

Figure 7.11: These graphs are organized like those of Figure 7.10. This time proprioception is used while learning, and the data from Tables 7.7 and 7.8 are used.

Comparing the learning-from-scratch and continual-learning results reveals no significant difference until Maze 9. As before, continual learning is superior in terms of training, but by less than in the non-proprioceptive case. In terms of testing, the margin is greater than before.

The clearest change over the non-proprioceptive case is the number of new units, which is higher for continual learning but lower for learning from scratch. The reason for this is not clear, but is most likely a combination of the following two factors. First there is less ambiguity in the proprioceptive case. Without proprioception, inputs are ambiguous both because the same sensory information is shared by several states, and because there are several ways of reaching each state. Proprioception resolves the second problem by identifying the way each state is entered. It also helps to resolve the first problem, since identification of the way a state was entered can also help identify what the current state might be. Entering a state labeled 4 from the north, for example, always means a move south is best (assuming the last move was the best), whereas entering it from the south means a move north is best. In the non-proprioceptive case, without knowledge of the last action, this determination would require previous sensory information, thus necessitating more units.

Second, the continual-learning case probably built extra units due to the non-optimal parameters. Nevertheless, the graph does seem to level off starting at Maze 5, indicating that those units already created were enough to learn the rest of the tasks. This leveling off also suggests the possibility that yet more challenging (but similar) mazes might require a small number of additional units.

The anomalously high value for the sixth maze while learning from scratch is probably due to poor parameter optimization for this maze, perhaps a result of using random seeds during parameter optimization that did not generate representative training samples.

What's interesting about the proprioceptive results is how well training transferred with continual learning in the later mazes. Very little extra training was generally required. In fact, Maze 8 never required any training in all 100 episodes, and in many cases the agent tested successfully on Mazes 5–9 after having been trained only on Mazes 1–4. More interesting is that the proprioceptive results are not greatly better than the non-proprioceptive ones. This indicates the ability of the agent to deal with highly ambiguous information that evolves and improves over time (as the agent's path becomes more consistent).

7.3.3 Hierarchy Construction in the Maze Environments.

It is interesting to examine what CHILD learns: when and which new units are built, and what hierarchies are constructed during the continual-learning process. In an actual example of continual learning from the set of 100 reported in the first (non-proprioceptive) case above, the following progression occurred during training. For the first maze, only one unit was constructed: $l^{20}_{\mathrm{MW},12}$. This unit resolved the ambiguity of the two maze positions labeled 12. With the unit in place, the agent learned to move east in the position labeled 10. It then could move north in position 12 if it had seen a 10 in the previous step, but west if it had seen a 6. Thus, the weight from s^6 to $l^{20}_{\mathrm{MW},12}$ was positive, and the weight from s^{10} to $l^{20}_{\mathrm{MW},12}$ was negative. The weight from s^0 to $l^{20}_{\mathrm{MW},12}$ was also negative, reflecting the fact that, though the optimal route does not involve moving south from the position labeled 0, during training this would often happen anyway. For the same reason, there was also a

negative weight from s^{12} to $l^{20}_{\mathrm{MW},12}$. If the agent was placed in one of the two positions labeled 12 at the beginning of the trial, it would move north, since it had no way of distinguishing its state. (It would then move south if it saw a 6 or east if it saw a 0.)

The second maze contains two new ambiguities: the positions labeled 9. Two new units were created: $l^{21}_{\mathrm{MW},9}$, and $l^{22}_{\mathrm{MN},12}$. The first, $l^{21}_{\mathrm{MW},9}$, clearly, was needed to disambiguate the two 9 positions. It had a strong positive weight from s^{12}. The second, $l^{22}_{\mathrm{MN},12}$, complemented $l^{20}_{\mathrm{MW},12}$. It was apparently needed during training to produce more accurate Q-values when the new 9 position was introduced.

The third maze introduces another position labeled 9. This caused a strong weight to develop from s^9 to $l^{21}_{\mathrm{MW},9}$. Only one new unit was constructed, however: $l^{23}_{20,12}$. The usefulness of this unit was minimal: it only had an effect if in a position labeled 12, the agent moved south or east. As a consequence, the unit had only a single weak negative connection from s^{10}.

The fourth maze introduces new ambiguities with the additional positions labeled 0 and 9 and two new labels, 1 and 8. No new units were constructed for this maze. Only the weights were modified.

The fifth maze introduces the additional ambiguities labeled 0 and 4. The label 4 definitely requires disambiguation, since the agent should choose different actions (MN and MS) in the two positions with this label. Since the agent can still move to the goal optimally by moving east in all positions labeled 0, no unit to disambiguate this label is necessary. Five new units were created: $l^{24}_{\mathrm{MS},4}, l^{25}_{21,9}, l^{26}_{25,9}, l^{27}_{26,9}$, and $l^{28}_{\mathrm{ME},9}$. The first disambiguates the positions labeled 4. It has a positive weight from s^9 and negative weights from s^0, s^4 and s^{12}. The second, third, and fourth new units, $l^{25}_{21,9}, l^{26}_{25,9}$, and $l^{27}_{26,9}$ all serve to predict the Q-values more accurately in the states labeled 9. The last new unit, $l^{28}_{\mathrm{ME},9}$, also helps nail down these Q-values and that of the upper position labeled 9.

The above example was one of the agent's more fortunate, though quite typical training runs. After learning Maze 5, CHILD tested perfectly in the remaining mazes without further training.

7.3.4 Non-Catastrophic Forgetting

Continual-Learning is a process of growth. Growth implies a large degree of change and improvement of skills, but it also implies a certain amount of skill retention. There are some skills that we undoubtedly lose as we develop abilities that replace them (how to crawl efficiently, for example, or, in general, how to act like a beginner once one has become a master). One would assume, however, that these abilities could for the most part be regained with less effort than they had originally demanded. This often tends not to be the case with neural networks in general, but it does seem to be the case with CHILD.

To test its ability to relearn long forgotten skills, CHILD was trained 100 times on the nine mazes (using continual learning without proprioception). Each time, after learning the last maze it was transferred back again to the first. The average number of training steps it needed to relearn Maze 1 was about 20% of the number it needed when it had had no previous training. The network built on average less than one unit to do this. However, its average testing performance was 20% worse than when the maze was first learned. Given the large difference in size and complexity between the first and last maze, it is quite surprising that in fully two-thirds of the cases, no retraining of any kind was required.

7.3.5 Distributed Senses

It was pointed out previously that because of its lack of traditional hidden units the Temporal Transition Hierarchies network could not make non-linear discriminations of its current input data. This is not necessarily a problem, however, since it can make such distinctions if the current data is repeated over several time steps. In reinforcement environments, this can be accomplished by giving the agent a "stay" action that leaves it in the same state. The agent can then stay in the same position for several time steps until it has made the necessary non-linear discrimination of its input data. This is a satisfying solution, since it more accurately simulates the temporal *process* of perception than does the traditional one-step, feed-forward mapping of neural networks.

This method worked in the mazes shown above. The sense vector consisted of five units: bias, WN (wall north), WW (wall west), WE (wall east), and WS (wall south). The bias unit was always on (i.e., had a value of 1.0). The other units had a value of 1.0 if there was a wall immediately to the corresponding direction of the agent; otherwise, they had a value of 0.0. For example, in positions labeled 12, the bias, WE, and WS units were on; in positions labeled 7, the bias, WN, WW, and WE units were on, etc. The agent was able to learn the first maze using distributed senses, but the cost was much higher training times than with local senses (averaging nearly 6000 steps).

It turned out, however, that the "stay" action was not needed and was rarely used, since the agent had other methods at its disposal. Removing the stay action still allowed the agent to learn the mazes effectively. In all positions except those labeled 0, the effect of the stay action could be achieved simply by moving into a barrier. (The 0 position on the other hand is uniquely labeled as the only state where the bias unit is the only active sensory unit.)

Furthermore, the agent can often make the necessary discriminations simply by using the context of its previous senses. Whenever it moves into a new state, information from the previous state can be used for disambiguation (just as in the locally encoded cases above). Now, however, previous sensory information is used to disambiguate states that are in principle unambiguous, but which are in practice difficult to discriminate. This surprising result shows that CHILD was able to learn to disambiguate its senses to only the extent needed to solve the task.

After the agent learned the first maze, it was transferred to the second in the same continual-learning process as described above. One outcome was that CHILD was able to generalize far better, and in some training episodes was able to solve *all* the mazes after being trained on only *the first two*! In order to do this it learned a modified right-hand rule, where it learned to follow the border of the maze around in a clockwise direction until it reached the goal; or, if it first hit a state labeled 0, it would instead move directly east. This is especially interesting when one considers the agent's primitive action apparatus. Had the agent been given a set of actuators that allowed it to move in a "forward" direction and then to turn when needed, such wall-following behavior would be less surprising. Though CHILD's action apparatus recognizes no distinction between its four sides, its behavior as it negotiates the walls gives one the distinct impression that it is moving forward and turning where appropriate. This seems in essence to be the effect of the hierarchies that CHILD evolved: a more resilient action apparatus. In one case, CHILD did this having created

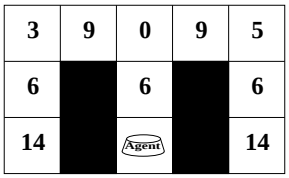

Figure 7.12: This environment was introduced by McCallum [59]. It works just like the maze environments of the previous sections with two exceptions: (1) if the agent attempts to move into a barrier, it receives a reinforcement of -1.0, and (2) the agent receives a reinforcement of -0.1 for all other actions that do not lead to the goal. As before, the agent receives a reward of 1.0 for taking the action that leads to the goal.

only six higher-level units. CHILD also often learned more direct, close to optimal routes to the goal, but the cost was the creation of more units (usually 15–20).

Generalization was clearly better with the distributed approach. There was a price, however. Frequently, solutions were not found to all the mazes, even after 1000 trials. This was in part because the parameters were not optimized for these simulations, but mostly it was due to the additional complexity of the task.

7.3.6 Other Reinforcement-Learning Methods

There are other methods that have been used for reinforcement learning in non-Markovian domains. One of these, McCallum's Utile Distinction Memory (UDM) [59] is based on Chrisman's Perceptual Distinctions Approach (PDA) [16], which is a constructive Hidden Markov Model (HMM) method. Both approaches build units to represent states explicitly. Both use traditional HMM training methods (i.e., the Baum-Welch procedure [78]) to estimate state occupation probabilities and to adjust observation and state-transition probabilities, and both combine these methods with Q-learning to learn action-utility values and to choose actions.

Results are given by McCallum [59] for his method in the environment shown in Figure 7.12. The Utile Distinction Memory approach consistently learned the maze in 2500 training steps. Transition hierarchies learned this task in an average of under 500 steps, indicating a factor of five speedup on this particular task. The parameters used were: $\eta = \beta = 0.6$, $\eta_L = 0.47$, $\sigma = 0.16$, $\Theta = 0.28$, $\epsilon = 0.2$, $\gamma = 0.74$, $\Delta T = 1.9$.

It's difficult to say how the two methods would compare on more complex tasks, though UDM scales approximately with the square of the number of states. In larger state spaces, such as Maze 9 above, transition hierarchies can make do with a small number of units — just enough to disambiguate which action to take in each state — whereas the HMM approaches need to represent most or all of the actual states. (Approximately two units were created on average to learn this task. UDM, in contrast, created between six and nine state units.) Also, both UDM and PDA must label every combination of sensory inputs uniquely (i.e., senses must be locally encoded). This greatly limits generalization such as was shown by Temporal Transition Hierarchies in the previous section.

Chapter 8

Synopsis, Discussion, and Conclusions

This chapter ties together the ideas raised throughout the dissertation. It begins with a discussion and interpretation of the results from Chapter 7. Next is an analysis of the contributions and deficiencies of CHILD and Temporal Transition Hierarchies. It closes with proposals for future work and some final thoughts.

8.1 Discussion of Results

Temporal Transition Hierarchies are clearly very fast compared to the networks measured against them in Section 7.2. In terms of the number of training patterns seen, they exhibit a speedup of more than two orders of magnitude. Considering the simplicity of the learning algorithm and its demonstrated speed (Section 7.2.1), the difference in computation time is probably even greater. It might be argued that some of the other algorithms in the comparison are more powerful, not being limited to Markov-k tasks. However, Markov-k tasks are interesting and important. Solving these quickly is necessary for dealing with many real-world problems, and such a sizable difference in speed gives one a certain degree of confidence in the general transition-hierarchy approach. It might also be argued that judging the performance of an algorithm requires more than two comparisons. Undoubtedly, more will be learned with further testing; however, the tasks used in the supervised-learning section of Chapter 7 cover a broad range of difficulties. One task deals with large numbers of context-sensitive events, one with small. One is stochastic, the other deterministic. One deals with large values of k (large time delays), and the other with small values. The fact that the same parameters were used across all values of k (in the *gap* task) reveals the stability of the algorithm and the reliability of the results. And finally, it's hard to ignore a two-orders-of-magnitude improvement.

After demonstrating the power of transition hierarchies, it remained to be shown that (1) the algorithm could be used for continual learning, and (2) that continual learning is a useful technique. Fortunately, these could both be done with a single set of tests. The tests compared the algorithm against itself, in one case with continual learning, and in the other case without. Temporal Transition Hierarchies were combined with Q-learning to produce a reinforcement-learning agent, CHILD. The agent was trained in several different environments many times and its performance was measured and averaged. The results proved that CHILD could learn complicated reinforcement-learning tasks quickly.

Continual learning was tested by first training CHILD on a simple task before training it on the next-most complicated one. A progression of nine such tasks was employed. By the end, the continual-learning case showed substantially better performance in the complicated environments than when no previous training had been done. There was no sign of catastrophic interference from earlier learning. In fact, the agent was able to return to the very first task of the series after having progressed to the last, and in two-thirds of the cases was still able to solve the task without retraining. When retraining was required,

substantially less was needed than when the task had first been learned. Taking an algorithm with already excellent performance and greatly improving it through continual learning shows two things. It shows that the algorithm is capable of taking advantage of and building onto earlier training, and it shows the usefulness of continual learning in general.

It would be interesting to test other amenable learning algorithms on the continual-learning environments of Section 7.3 and measure to what extent they are helped or hindered by earlier training. Nevertheless, comparing CHILD against itself to prove the feasibility of continual learning seems well-justified for several reasons. First, the system allows incremental hierarchical development, which is critical to continual learning. Second, it handles reinforcement-learning problems with ease, since it can be used for predicting continuous values in noisy domains. Third, it's very fast. With a slower algorithm, a good comparison might not be computationally feasible.

Given an appropriate underlying algorithm, and a well-developed sequence of environments, the effort spent on early training more than pays for itself later on. But continual learning is not just a means of shaping an agent toward solving a difficult task. It is motivation for training an intelligent agent when there is no final goal or task to be learned. If an agent is trained for a particular task, its abilities may someday need to be extended to include further details and nuances. For example, the agent might initially be trained only up to the eighth maze of Figure 7.9 (its training thought to be complete), but later the trainer might need an agent that can solve the ninth maze as well. Perhaps there will continue to be more difficult tasks after the ninth one too.

One anticipated result was that the agent developed skills helpful to its later learning. Before training began, what those skills would be was not known. One such skill was a small dance that the agent always performed upon beginning a maze in certain ambiguous positions. The dance is necessary since in these ambiguous states, neither the current state nor the correct actions can be determined. Once an agent performed the dance, it would move directly to the goal. Sometimes the dance was very general, and it worked just as well in the later mazes as in the earlier ones. In fact, it was very common for the agents to learn by Maze 4 the behaviors necessary to solve all remaining mazes without further training. Frequently, however, the skills did not generalize so well, and the agent had to be trained in each of the later environments. Before training began, however, it would have been extremely difficult for the trainer to know which skills would be needed. In any real-world task, it will be even more difficult to know beforehand which skills the agent will need. This is precisely why continual-learning is necessary — to remove the burden of such decisions from the concerns of the programmer/designer/trainer.

8.2 Deficiencies

Most of the weaknesses of the Temporal Transition Hierarchy approach were discussed in Section 6.3. In particular, the version of the algorithm used to produce the results is poor at classifying patterns from a single time step. In order to classify complicated patterns, it must enlist higher-order units. However, these units only receive information from *previous* time steps and help little in the classification of the current time step. The only way the agent can classify a complicated sensory pattern is by staying in the same position for

several time steps. This turned out not to be a particularly serious problem, however, and agents were in fact able to learn to do exactly that.

A second problem is that the original version of the algorithm can only solve Markov-k tasks. The system cannot remember events for any arbitrary duration, but only for specific, already-learned durations. This means that the network could not learn, for example, an *embedded* Reber grammar (a grammar in which the Reber grammar appears as a subcomponent [17], and which requires information to be remembered for an indefinite duration). A variation on the algorithm that could conceivably solve this problem is discussed in the future-work section, Section 8.4.

Another deficiency of transition hierarchies is their proclivity for creating new units. When the parameters are properly set, usually only the needed units are created. In some circumstances, particularly with poor parameter settings, more units can be created than are really necessary. The extra units slow down the operation of the system. This would not be a particularly serious problem if there were a way to get rid of those that are not needed. This has proven to be fairly difficult, however.

Besides the fact that it is based on Temporal Transition Hierarchies which are limited to Markov-k environments, CHILD has other weaknesses — due to its use of Q-learning. One of the limitations of Q-learning, or any other of the reinforcement-learning methods based on dynamic programming is that it requires a fixed number of actions. Continuous actions are still not generally feasible. Another problem with reinforcement-learning methods based on dynamic-programming is their assumption of a fixed reinforcement landscape. If the reinforcements in the environment changes, the computations must be done anew. Ways of circumventing this problem are discussed in Section 8.4.

8.3 Contributions

CHILD has taken a successful first step towards continual learning. The problems it solves are difficult; it solves them quickly; and it solves them even more quickly with continual learning. The strengths of Temporal Transition Hierarchies make this possible. Among these strengths is their ability to handle a large degree of stochasticity in early stages of reinforcement learning and again whenever a new environment is introduced. (When beginning in a new environment the temperature is reset to 1.0, and new Q-values must be learned because the Q-values do not transfer, even though the policy based on them might.)

A second strength is that learning is the same at all levels of the hierarchy. There is no distinction between learning complex behaviors and learning simple ones, or between learning a new behavior and subtly amending an existing one. This is what is meant by hierarchical development: New behaviors are composed from variations on old ones (as was seen in Section 7.3.3). A third strength is the fact that the algorithm is incremental: it does not need a fixed set of data from which to learn but can learn from the environment in whatever order data is encountered.

Another strength of the algorithm is that in general, only skills needed for the task are acquired. New units are added only for strongly oscillating weights. Since the network is learning to predict Q-values, large oscillations in the weights reflect large differences in Q-value predictions, which reflect large differences in potential reward. Predictions vary the most in states where prediction improvements lead to the largest performance gains.

And this is exactly where new units will first be built. When weights are not varying much, this indicates that predictions are close to correct or that the Q-values are small; these are places where improvements in prediction have little impact on performance, and where new units are less likely to be created. (This approach to unit creation is done more explicitly by McCallum [59]). The new units extend the agent's current repertoire of skills by modifying its behavior (i.e., revising the Q-values on which its policy is based) to take advantage of current contextual information. The contextual information, furthermore, can eventually extend into the past for any arbitrary duration, creating behaviors that can also last for any arbitrary duration (another strength).

8.3.1 Distributed Hierarchical Control

One of CHILD's most important strengths is the distributed nature of its hierarchical control. At its inception, a complex behavior is encoded into the units of a Temporal Transition Hierarchy network as a distributed pattern of activation. The pattern specifies a plan of future activity where all anticipated contingencies are specified in advance. Each high-level unit is a link in that plan and carries the agent's intentions through time for cases where the exact sequence of future events is not certain. Take, for example, a behavior such as the following (left column) that might be activated by the input "0" in Maze 9 (Figure 7.9).

`Move east.`	ME,
`If the input there is "0",`	
`  move east again.`	< 0, ME $>$,
`If the input is "4", however,`	
`  move north, then`	< 4, MN $>$,
`  if the next input is "4",`	
`  move north again,`	$< 4, < 4$, MN $>>$,
`  but if the input is "5",`	
`  move south and then`	$< 4, < 5$, MS $>$,
`  move south again upon seeing "4".`	$< 4, < 5, < 4$, MS$>>>$

This behavior can be encoded by a *set* of activations across all appropriate units, such as the units for "move east", "move east after sensing 0", etc., as shown in the second column above. (In order to correctly predict Q-values, however, these units must not simply be activated, but be activated to a specific *degree*.) Besides these, units that conflict with the intended behavior must get zero or negative activation.

These kinds of "rules" are constructed during learning through the creation of new units. After a unit is built, its weights attempt to learn the rule's applicability (i.e., under what conditions and to what degree the rule should be used).

A behavior ends when an input is not anticipated and the appropriate action is unclear, i.e., when the agent has no most-highly activated action unit. These occasions are opportunities for extending the just-completed behavior. Extending a behavior does not necessarily require a new unit. If a behavior is viewed as a policy — a mapping from specific inputs to specific responses — extending a behavior can be done simply by learning a new policy move at the point where the next action is unclear.

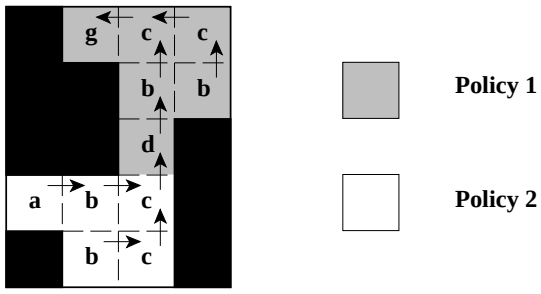

Figure 8.1: In order to reach the goal (g) the agent must have a different policy in the shaded region from that in the unshaded region. Letters represent sensory input, and arrows indicate optimal policy moves.

So when is a new unit needed? When the agent's input is ambiguous and no single policy move for that input is optimal. This situation requires a *context-sensitive policy*, and Temporal Transition Hierarchies provide a hierarchical, context-sensitive policy. This is perhaps CHILD's strongest contribution. Each level of the hierarchy enables a different policy in different contexts. In the environment shown in Figure 8.1, the agent needs a different policy in the shaded region (Policy 1) from that in the unshaded region (Policy 2). In particular, in the shaded region the input "b" should prompt the agent to move north, and "c" should prompt it to move west; whereas in the unshaded region, "b" demands a move east, and "c" demands a move north. A hierarchical, context-sensitive policy would activate Policy 1 when "a" was encountered and keep it active until "d" was encountered. At that point, Policy 2 would be activated until the goal, "g," was reached. This would be specified at the second level of the hierarchy as a new higher-level policy: "a $\rightarrow$ Policy 1, d $\rightarrow$ Policy 2". Still higher-level policies could be formed, if, for example, "a" in some contexts invoked Policy 1 but in other contexts invoked a different policy, Policy 3.

Context-sensitive policies as just described are very similar to Dawkins' "hierarchy of decisions" (Section 1.4.1). Temporal Transition Hierarchies do nearly exactly this, with a couple of exceptions. First, with Temporal Transition Hierarchies, a context-sensitive policy is in effect for only a fixed number of time steps. If the agent gets stuck in the same region for too long, it will forget which policy it should be following (a weakness that seems to occur in humans too). A possible solution to this problem, recurrent transition hierarchies, is suggested in Section 8.4. Second, the difference between Policy 1 and Policy 3 might be only a single response to a single stimulus. Temporal Transition Hierarchies focus on only the differences, never forming concrete concepts for Policy 1 or Policy 2; which are instead encoded as patterns of activation across the higher-level units. This distributed approach of transition hierarchies is a far more efficient way of encoding context-sensitive policies than forming an entirely new policy for every tiny difference.

Building context-sensitive, hierarchical policies automatically while learning continuous, context-sensitive Q-values is an approach quite different from any hierarchical reinforcement-learning method proposed previously. The two greatest distinctions are: (1) it is context sensitive and can handle ambiguous perceptual information, and (2) the hierarchies are constructed automatically in a bottom-up fashion, instead of top-down and by hand. This is in marked contrast to the approaches of Dayan and Hinton [21], Jameson [42], Kaelbling [47],

	Dimension of Complexity	CHILD
1	Sense/Action Representation	Distributed
2	Sense/Action Values	Continuous
3	Sense→Action Mapping	Linearly-separable
4	Sense→State Mapping	Many-many
5	State→Action Mapping	Many-many
6	Next State Function i.e., (state, action)→state	Stochastic (Many-many)
7	Underlying Model	Markov-k
8	History Information Kept	Any Fixed Amount
9	Duration History is Kept	Fixed
10	State→Reinforcement Mapping	Many-one
11	Reinforcement Planning Steps	Any Positive Value

Table 8.1: The eleven dimensions of difficulty from Table 2.1 and how CHILD measures up.

Lin [55], Singh [100], and Wixson [126]. These methods also combine multiple policies, but they require complete state information and architectures pre-designed for a particular task.

In all hierarchical architectures there is the issue of "vertical" credit assignment: assigning credit to the correct level of the hierarchy. In most systems this can be a very tricky issue. In Temporal Transition Hierarchies it occurs automatically as part of the learning process. Assignment of credit is done by calculating the gradient of the error with respect to the weights, no matter where in the hierarchy they lie. Since standard reinforcement-learning methods already solve the difficult task of transforming reinforcement signals into error signals, vertical credit assignment in reinforcement-learning environments is straightforward.

8.3.2 Rating CHILD with the Dimensions of Difficulty

Using the dimensions enlisted in previous chapters to measure the difficulty of environments and the capabilities of learning algorithms, CHILD is described in Table 8.1 in terms of the environments it can learn. CHILD mostly has limitations along Dimensions 3, 7, and 9. Some of these limitations may be alleviated by possible modifications mentioned in the next section.

Not all attributes of a continual-learning agent are readily apparent from the table. Among these, signified by Dimension 8, is the need for continuous growth in the amount of history information the agent can keep. Also not obvious from the table is the fact that continual learning should not be limited to finite environments where exact situations are repeatable, but should instead participate in non-repeatable, real-world environments where no regularities can be absolutely guaranteed.

8.4 Future Work

CHILD provides a strong foundation for continual learning, but it does not address every issue that may be required. Some extensions are immediately obvious. Two of these are (1) the use of transition hierarchies in non-temporal domains and (2) the addition of

recurrent connections. These will be explored in the following two subsections. After these discussions come some other extensions eagerly waiting to be addressed.

8.4.1 Stationary Mappings

Just as Temporal Transition Hierarchies can make non-linear discriminations if an input is repeated over multiple time steps, the transition-hierarchy approach can be extended to non-sequential data. A purely non-temporal, feed-forward network can be constructed by removing time dependencies from the algorithm of Section 6.2.5. The derivation of a learning rule for this variation is given in Appendix D. This derivation is nearly identical to that of Section 6.2.3. The major difference is that computation of the unit's activation values must begin at the highest unit and work top-down instead of bottom-up. This is necessary since the high-level units would otherwise have no effect on lower units. Without time dependencies, the algorithm strongly resembles that of Sanger [89] and Sanger et al. [90].

8.4.2 Recurrent Connections

One obvious drawback of Temporal Transition Hierarchies is that they can only learn Markov-k tasks (learning k in the process; see Section 2.2.3). One problem mentioned over the previous chapters has been the arbitrary-duration problem: how a piece of information can be kept for an unspecified period of time. Temporal Transition Hierarchies always keep information for a specific and precise period of time. (Unit n^i holds information that will have a discernible effect on the outputs of the network exactly τ^i time steps into the future.) One simple way of overcoming this limitation is to allow high-level units to have output activation values. That is, the network could be made fully connected, so that higher-level units have outgoing weights to the rest of the network and thus serve as hidden units as well as higher-order connections. High-level units could then make use of their own previous values as well as the values of other high-level units. These new *recurrent* connections could in principle allow information to be stored by the network for as long as needed.

The learning algorithm would have to be modified to make use of the recurrent connections, but it is conceivable that the network might form long-lasting representations even without such modifications. The networks proposed by Jordan [43] and Elman [25], for example, are capable of learning to store information over several time steps without performing gradient descent on their recurrent connections (Section 4.3.3). Unlike these networks, however, the units of transition hierarchies have specific roles in that they must generate exact values that reduce the error of their respective connections. In the process of doing this, they may learn to detect features that can then serve as reliable inputs to other units of the network, including themselves.

Though it seems inappropriate to do complete gradient descent with respect to recurrent connections in the higher-order system presented above — which would cause each unit to share responsibilities as a higher-order connection and as a hidden unit — there is an agreeable compromise. The problem addressed by the recurrent connections is that of allowing a higher-level unit (and thus the weight it modifies) to retain its value for an arbitrary length of time. This, however, can be solved without doing complete gradient descent in a fully-connected recurrent network. Instead, the network could have just a

single recurrent connection from each high-level unit to itself and then perform gradient descent with respect to these connections. This is the same elegant approach used by Bachrach [4] and by Mozer [65] (Section 4.3.1). Using Bachrach's approach, these units can keep their values for any arbitrary duration. This enables each high-level unit to tune in to precisely those signals that should cause it to change its activation value, regardless of when those signals occur. It can then keep that value for as long as necessary.

The derivation of the learning rule for transition hierarchies with single self-recurrent connections is somewhat different from that of the non-recurrent system. Because the units have self-connections and can retain information without explicit use of time delays, unit activations can be computed top-down as with the stationary network (highest l units first, down through lower-level l units, terminating with the output units). This means that all weights contribute to the output at every time step, which in turn implies that, just like with the stationary network above, non-linear discriminations could be made in a single forward propagation. The derivation of a learning rule for this kind of architecture is given in Appendix E.

Recurrent connections might extend Temporal Transition Hierarchies in useful ways, but even more powerful memory extensions will eventually be needed — a dynamically accessible short-term memory, for example. Some everyday tasks often require the temporary memorization of a few basically random pieces of information, such as one's shopping list, or what the next few words in one's sentence will be. Once such information is used it can immediately be forgotten. This kind of memorization can be implemented using some kind of push-down automaton. Neural-network algorithms that learn to use a stack [67, 101] do exist. Integrating these into a reinforcement learning environment would be a good next step. Making them amenable to continual learning will be more difficult.

8.4.3 The Changing-Reward Problem

Another problem faced by CHILD in Chapter 7 was that the environments kept changing. The tests were implemented this way to replicate aspects of *shaping*. Shaping is the technique of training an animal to perform a complex skill through successive approximation (very much like continual learning except that there is always a definite, specific, final behavior to be learned). A pigeon might be trained, for example, to peck Button A by rewarding it whenever it crosses into the half of its cage where Button A is located. After it has learned to stay in that half of the cage, the reward area could be reduced from one half to one quarter, etc., until the pigeon stays near Button A. Rewards could then be given whenever the pigeon's beak moves toward the button, and so on. This requires constant modification of the reward landscape, however, which means that a pigeon implemented with Q-learning will be re-learning most or all of its Q-values again and again.

One possible solution to the problem of a changing reinforcement landscape is simply not to use Q-learning. If the AHC were used instead (Section 5.1), then the agent could transfer its skills to a similar domain easily. The critic module might need quite a bit of modification while the policy module might need very little. If the policy values were to change slowly, but the critic values changed quickly, then the critic could be straightened out before the policy had been changed much. Another possible solution is to pursue the notion introduced in Section 5.5: the *absolute reinforcement elevation* (the "goodness" of each state) could be separated from the *relative elevation* (the difference in "goodness"

between a state and its neighbors). The absolute elevation values would still need to be relearned whenever the environment or reward position changed slightly, but most of the relative values would not. The last absolute-elevation prediction could also be supplied as an extra input. Its new value could then be calculated from its last value and from the relative elevation values (instead of being re-calculated at every time step from sensory input alone). Defining policies in terms of relative elevation allows skills to be more easily transferred to environments with slightly different rewards (and therefore slightly different absolute reinforcement elevations).

There are two alternative approaches to shaping, however, that are less confusing to the pigeon. One would be to start the pigeon with its beak right next to Button A, rewarding it when it inevitably hit the button, and then moving it successively farther away. This has the advantage of keeping the environment and reinforcement landscape more or less the same (except for the person holding the pigeon). It may not always be effective, though, as can be seen from Maze 9 in Figure 7.9. Had the agent first learned that it should absolutely always move west given an input of "9," it would have had great difficulty learning to move east given an input of "9" when it finally reached the two doorways.

The final and probably best alternative to the changing-rewards problem is just to come up with a single, intelligent reinforcement landscape and then to leave it alone! What's needed is a smart, layered reinforcement scheme. Here's an example. Let's say a mouse should learn to walk around a maze in search of cheese. Assuming the agent has real and not simulated legs and no knowledge of its sensors or effectors, it needs first to learn to move its limbs in a controlled way. It should therefore be given a reinforcement when it does so. Once it has learned to move its legs about without damaging itself, all such movements will result in equal reward and will therefore become equally probable. If organized movements of its legs at some point result in the agent moving forward, then it will receive extra reinforcement *in addition to* what it receives for the controlled limb movements. A large set of "good" action sequences will be narrowed down to a smaller set of "better" sequences without modifying the reinforcement landscape. As the agent gets better and better at moving forward, it will get a higher density of reward per action and will eventually be moving around constantly. Sooner or later it will find some cheese, and it will get a really big reinforcement. Whenever the agent discovers a source of reinforcement higher than it is used to, it will modify its behavior, increasing its average reinforcement to a new level until, adept at the new behaviors and used to the new level, it stumbles upon a still higher reinforcement. This method fits in best with the philosophy of continual learning in that there need not be a maximum achievable reinforcement — continual improvement leads indefinitely toward greater average reward.

The process of continually increasing the reinforcement level as the complexity of the behavior increases is quite satisfying and seems very realistic. It requires access to an intelligently constructed reinforcement landscape, however, which may be hard to supply. Perhaps the agent might eventually be allowed to construct its own system of short-term rewards in order to better teach itself skills that will bring it greater long-term reinforcement [20].

Even the layered-reinforcement method is limited, though, because it assumes a single (though possibly very complicated) task. This is the case in shaping, but since it is not the case in continual learning, a different approach may be needed. One option is to train the agent to seek different goals in different contexts, i.e., to develop a different context-sensitive

policy for each goal, provided the goal is specified in advance (like the tasks considered by Singh [99]). Another very promising option would be to use Temporal Transition Hierarchies to learn a *model* of the environment (Section 5.4), inverting the model to choose good actions or to train a *controller*. This is advantageous in that a model maintains information about the environment that is equally valuable when the goals change. Also, a model and controller work well together in continuous domains, which is otherwise difficult for Q-learning. Learning a model of the environment still involves learning the reinforcement landscape, however, no matter how temporary that landscape might be, and a new controller must be trained whenever the landscape changes.

A method that avoids retraining with changing goals is to train on all possible goals simultaneously (cf. Kaelbling [47, 48]). This method is highly promising in that it merges controller and model together into an expanded model of the environment. Models usually represent mappings of the form:

$$(\text{state, action}) \rightarrow \text{next-state.}$$

Given the current state and intended action, the next state is calculated. This can be used for single-step lookahead or potentially for a deeper search, but is usually used for training a controller. An "all-goals" model, on the other hand, could learn mappings of the form:

$$(\text{state 1, state 2}) \rightarrow (\text{action, cost}),$$

mapping every pair of states to (1) the best initial action when moving from the first to the second, and (2) the expected cost of the journey. Learning this mapping can be done with a simple dynamic-programming-style algorithm that allows the agent to change goals without retraining. Finding the best action is computed instantly without an expensive search or constraint satisfaction process. The method's drawbacks are its $O(n^2)$ space requirement, and the fact that it needs full access to the agent's state at all times. The second requirement makes its conversion to non-Markovian environments challenging. Nevertheless, the potential payoff of an all-goals continual-learning algorithm in non-Markovian environments makes this research quite compelling.

8.4.4 Practical and Theoretical Work

Other important areas of future work range from the very practical to the very theoretical. On the practical side, a method needs to be developed that allows useless units to be pruned from the transition-hierarchy network. The difficult part, of course, is identifying which units are useless. This is more complicated than the problem of weight elimination in standard neural networks. A large hierarchy may be very vital to the agent, and the lower-level units of the hierarchy may be indispensable, but before the hierarchy is completed, the lower units may serve no purpose. How to identify these units as potentially useful seems a very tricky issue, particularly in stochastic domains.

Another of the practical issues is simply that of testing the network and its variations further. Many tasks would likely benefit from the transition hierarchy approach, such as time-series prediction, speech recognition, speech production, and other context-sensitive serial tasks.

On the theoretical front are two issues related to dynamic programming. The first is how learning a context-sensitive policy can be integrated into the standard dynamic-programming framework. Interest is rising among reinforcement-learning researchers in the area of partially observable Markov decision processes (POMDP's), which are already described by a broad theory of dynamic programming in hidden Markov environments. Applying this theory to transition hierarchies is a large but promising project.

Another, much larger issue related to dynamic programming is that of regularity and iteration. Existing reinforcement-learning theories do not apply to environments such as the real world that cannot be perfectly modeled or reset and iterated for a countless number of trials. A theory for measuring regularity and maximizing future reinforcement based on these regularities will someday need to be developed. There is still much groundwork that must first be laid for such an ambitious project, however.

8.5 Closing Thoughts

Continual learning is the learning methodology that makes the most sense in the long term. We know this clearly from our own experience. We do not begin our education by working on our dissertations. It takes (too) many years of training before even beginning a dissertation seems feasible. It seems equally unreasonable for our learning algorithms to learn the largest, most monumental tasks from scratch, rather than building up to them slowly.

Implicitly, we all understand the significance of continual learning. We do it without thought. It simply happens. Even when we continue the process in overt ways, we sometimes fail to see its presence in our actions. Our very acts of producing technology, building mechanisms, writing software, is to make automatic what we can only do slowly and with the agony of constant attention. The process of making our behaviors reflexive is onerous; one can in fact learn to do arithmetic as quickly as a calculator, but writing a program to do our calculations for us is much easier. We build our technology, particularly our software, as extensions of ourselves. Just as we develop skills, stop thinking about their details, and then use them as the foundation for new skills, we develop software, forget about how it works, and use it for building newer software. We do the thinking, then make our software do what we've figured out how to do automatically. Building robots to do our manual work may be more efficient than doing it ourselves, but we cannot continue to design specific solutions to every complicated, tiny problem. As efficient as programming may be in comparison with doing a task ourselves, it is nevertheless difficult, tedious, and time-consuming; and program modification is even worse. We need robots that learn tasks without specially designed software solutions. We need agents that learn and don't forget, keep learning, and continually modify themselves with newly discovered exceptions to old rules.

Continual learning makes shaping a reality, and goes beyond it. Continual learning is learning on top of learning. It is what we do when we learn to read, when we learn to write. When we learn to read music, we first learn what notes are, then their names, what they sound like, how to reproduce them; then we learn simple rhythms, simple sequences of musical intervals, more complicated rhythms, more complicated musical intervals, different keys, different clefs, different speeds, different notations, more and more complicated

sequences of intervals and rhythms, and on and on. And towards what end? Does anyone ever finish learning to read music? Do we finish learning how to write or do research? Do we ever learn anything *completely*? Or do we just keep getting better than we were before?

Appendix A

Simulating a Queue With a Focused Network

It is not difficult to devise an activation function that would allow a Bachrach [4] or Mozer [65] network to simulate a queue (i.e., to produce time-delayed output) given discrete inputs. One way to do this with *differentiable* transfer functions is as follows.

Without loss of generality, assume the input to each input unit is binary. Allow one hidden unit for every input unit. There will also, clearly, be the same number of output units as there are input units. The i^{th} input unit is s^i; the i^{th} hidden unit is h^i; and the i^{th} output unit is o^i. Use the following differentiable functions for each hidden unit, h^i:

$$\begin{aligned} h^i(t) &= f_H(\sum_j s^j(t) w^H_{ij} + d h^i(t-1)) \\ f_H(x) &= x - \frac{1}{2}\sin^2(\pi x), \end{aligned}$$

and use the following differentiable functions for each output unit, o^i:

$$\begin{aligned} o^i(t) &= f_O(\sum_j h^j(t) w^O_{ij}) \\ f_O(x) &= \sin^2(\pi x). \end{aligned}$$

Should the network have the following hidden-layer weights (w^H), output-layer weights (w^O), and self-connection (decay) weights (d), it will be able to store an arbitrary number, D, of input tokens into the hidden units to be retrieved D time-steps later by the output units.

$$\begin{aligned} d &= \frac{1}{2} \\ w^H_{ij} &= \begin{cases} 2^{D-1} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \\ w^O_{ij} &= \begin{cases} \frac{1}{2} & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

See Pollack [75, Ch. 4] for the clearly related technique that inspired this one.

Appendix B

Equivalence of SLUG and Second-order Recurrent Networks

The equivalence of SLUG and the other second-order recurrent networks presented in Section 4.3.4 becomes clear once SLUG is formalized. The following equation describes SLUG [5]:

$$\vec{o}(t) = \mathbf{W}_{a(t)}\vec{o}(t-1),$$

where $\vec{o}(t)$ is the vector of hidden-unit outputs at time t; $a(t)$ is the single action taken at time t; and $\mathbf{W}_{a(t)}$ is the weight matrix corresponding to action $a(t)$. Therefore,

$$o^i(t) = \sum_j w_{ij}(t)o^j(t-1), \tag{B.1}$$

where $w_{ij}(t)$ is the jth weight in the ith column of matrix $\mathbf{W}_{a(t)}$. Because only one action is taken at a time, the actions may be encoded locally as a unit vector, $\vec{a}(t)$, with the number of dimensions equal to the number of actions. A third-order tensor, $\hat{\mathbf{W}}$, can be constructed such that when it is multiplied by the vector $\vec{a}(t)$, it produces the matrix $\mathbf{W}_{a(t)}$.

$$\mathbf{W}_{a(t)} = \vec{a}(t)\hat{\mathbf{W}},$$

Let each element i, j, k of $\hat{\mathbf{W}}$ be written w_{ijk}, then

$$w_{ij}(t) = \sum_k w_{ijk}a^k(t).$$

Therefore, from B.1,

$$\begin{aligned} o^i(t) &= \sum_j \sum_k w_{ijk}a^k(t)o^j(t-1) \\ &= \sum_{j,k} w_{ijk}o^j(t-1)a^k(t). \end{aligned} \tag{B.2}$$

By rearranging $\hat{\mathbf{W}}$ into $\hat{\mathbf{W}}'$ as follows:

$$w'_{ijk} = w_{ikj} \quad \forall i, j, k,$$

Equation B.2 can be rewritten to be precisely Equation 4.6 (provided the function f is the identity map):

$$o^i(t) = \sum_{j,k} w'_{ijk}o^k(t-1)a^j(t).$$

Appendix C

Parameter Values for the Maze Tasks

The parameters for the learning-from-scratch maze simulations (Section 7.3.1) are plotted in Figure C.1 to show their changes as the mazes become more complex. It seems reasonable that γ should increase as the mazes grow larger, since this allows the reinforcement at the goal to spread more evenly back towards the most distant reaches of the maze. It also seems reasonable that σ should decrease while θ should increase as the mazes get larger. The effect is to impede the creation of new units. (Increasing ϵ will have the same effect.) As the mazes grow in size, more training needs to be done before it can be reliably established which connections are oscillating due to insufficient information. During early training, there will initially be a significant amount of noise due to changing Q-values even in states that are not ambiguous. Eventually, the Q-values will stabilize to their correct values wherever possible, and the remaining probabilistic effects noticed by the learning

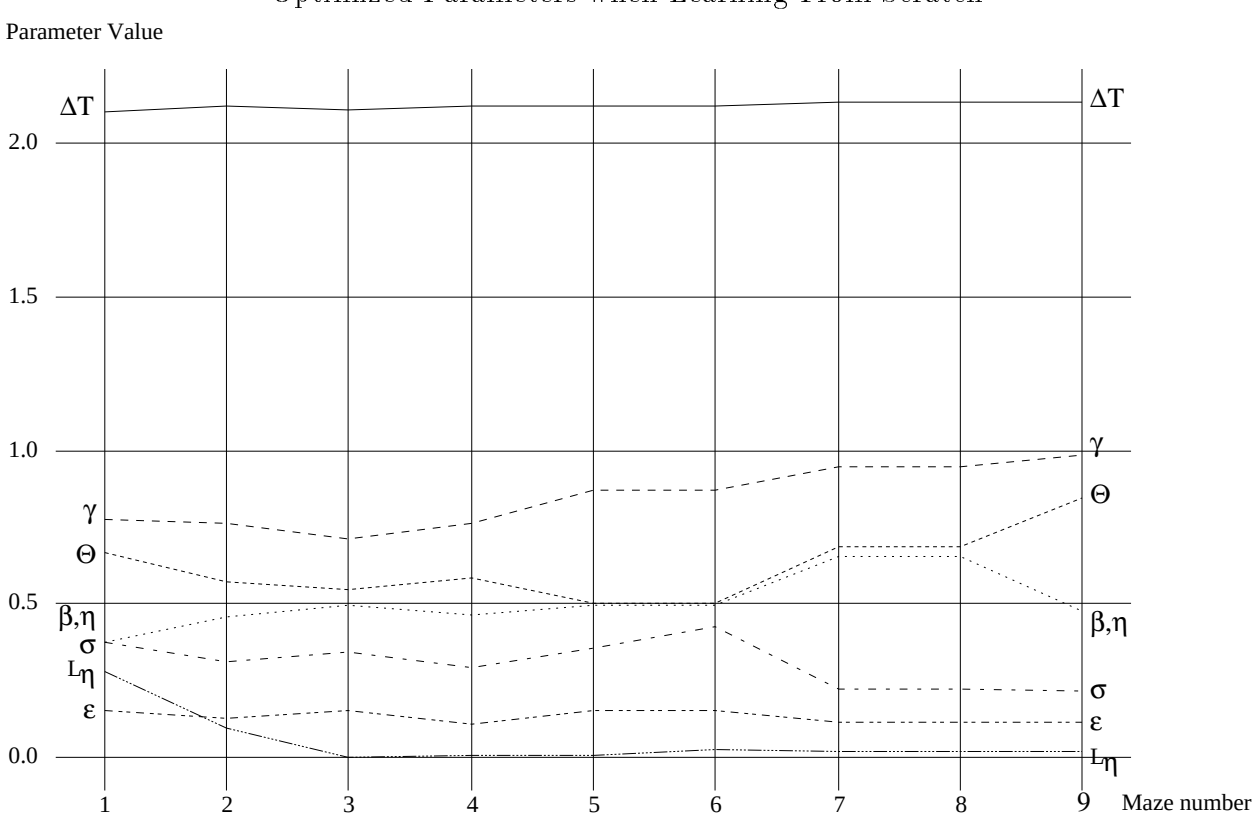

Figure C.1: All seven parameters for the nine mazes are shown with their optimized values. The horizontal axis shows the maze number. The vertical axis measures the value of each parameter.

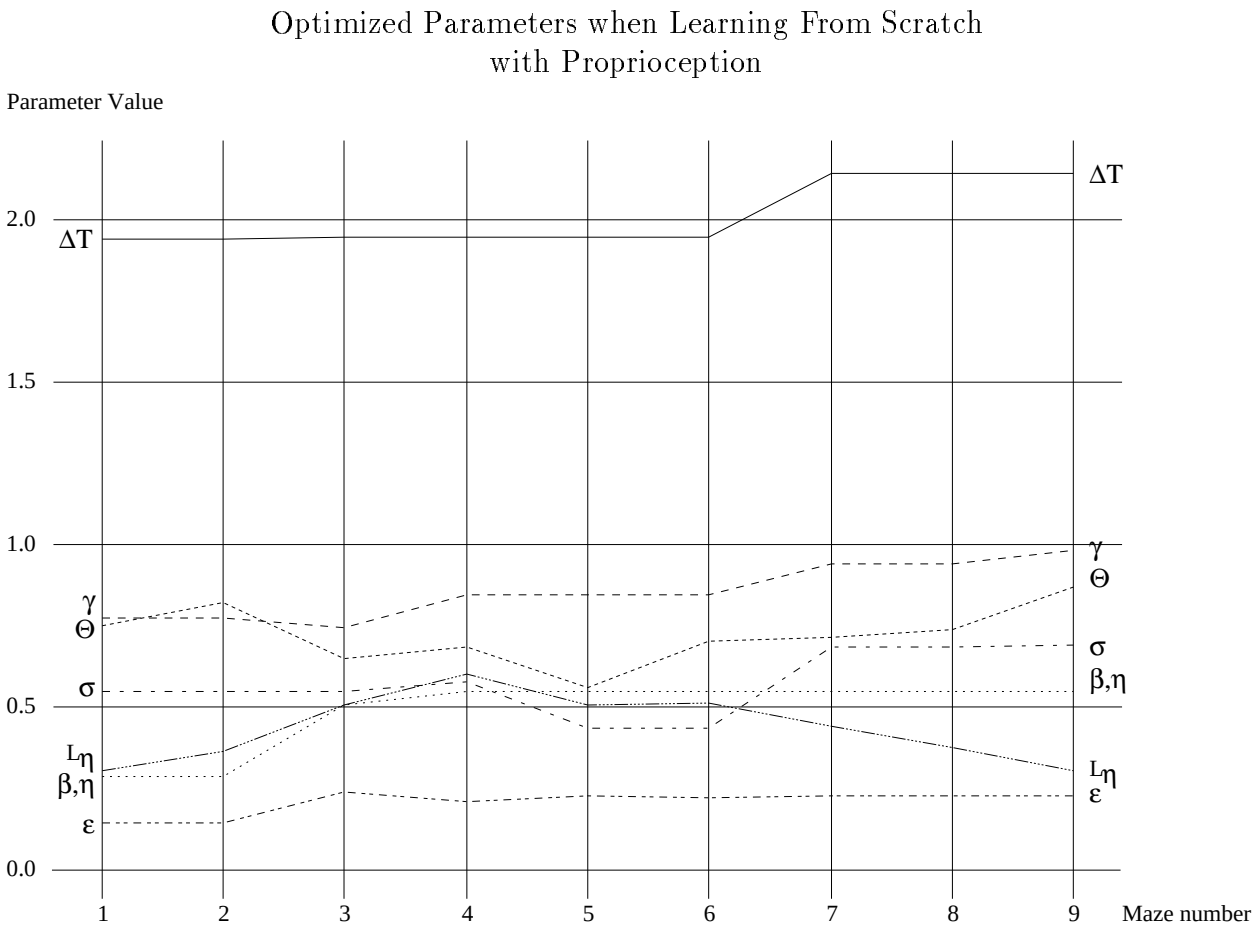

Figure C.2: All seven parameters for the nine mazes are shown with their optimized values when trained with proprioception. The horizontal axis labels the maze number. The vertical axis measures the value of each parameter.

algorithm will be due completely to state ambiguities. Therefore, in large and complex mazes — where there will initially be a great deal of unpredictable behavior — it is best for unit creation to occur slowly, allowing the weights to settle to their correct values wherever they can. In small, simple environments, aggressive unit creation may be acceptable, since the weights should converge fairly quickly to their optimal values anyway. The reasons that the creation of new units is apparently favored while training Maze 6 is not clear.

The parameters for the proprioceptive learning-from-scratch simulations of Section 7.3.2 are shown in Figure C.2. Drawing conclusions from this graph is not as straightforward as the previous case, though the trend in γ is the same.

Appendix D

Derivation of Learning Rule for Non-Temporal Network

The following derivation is nearly identical to that given in Section 6.2 but has been altered to remove the temporal component. It can therefore be used for static classification tasks, and very closely resembles the network of Sanger [89]. The equations below have nearly a one-to-one correspondence with (and are therefore numbered to correspond with) those of Section 6.2 and may be viewed side by side with them for greater clarity.

$$\begin{aligned} S &\stackrel{def}{=} \{u^i \,|\, 0 \le i < ns\} \\ A &\stackrel{def}{=} \{u^i \,|\, ns \le i < ns + na\} \\ L &\stackrel{def}{=} \{u^i \,|\, ns + na \le i < nu\} \\ N &\stackrel{def}{=} \{u^i \,|\, ns \le i < nu\} \end{aligned}$$

$$\begin{aligned} s^i &\in S \\ a^i &\in A \\ l^i_{xy} &\in L \\ n^i &\in N \end{aligned}$$

$$n_i = \sum_j \hat{w}_{ij} s^j \tag{D.3}$$

$$\hat{w}_{ij} = \begin{cases} w_{ij} + l_{ij} & \text{if a high-level unit } l_{ij} \text{ for weight } w_{ij} \text{ exists} \\ w_{ij} & \text{otherwise} \end{cases} \tag{D.4}$$

The total error is now the sum of the errors over all patterns in the training set. The index p is used to denote patterns from the training set:

$$E = \sum_p E^p.$$

In this derivation the error will be accumulated throughout a training epoch.

$$E^p = \frac{1}{2}\sum_i (T^{ip} - a^{ip})^2 \tag{D.5}$$

$$\Delta w^p_{ij} \stackrel{def}{=} \frac{\partial E^p}{\partial w_{ij}} \tag{D.6}$$

$$w_{ij} \leftarrow w_{ij} - \eta \sum_p \Delta w^p_{ij} \tag{D.7}$$

For the remainder of the derivation, it will be assumed that all variables are indexed with respect to p, the current pattern.

$$\Delta w_{ij} \stackrel{def}{=} \frac{\partial E}{\partial w_{ij}} \tag{D.8}$$

$$\delta^i \stackrel{def}{=} \frac{\partial E}{\partial n^i} \tag{D.10}$$

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}} &= \frac{\partial E}{\partial n^i}\frac{\partial n^i}{\partial w_{ij}} \\ &= \delta^i \frac{\partial n^i}{\partial w_{ij}} \end{aligned} \tag{D.11}$$

$$\begin{aligned} \frac{\partial n^i}{\partial w_{ij}} &= s^j \frac{\partial \hat{w}_{ij}}{\partial w_{ij}} \\ &= s^j \begin{cases} 1 + \frac{\partial l_{ij}}{\partial w_{ij}} & \text{if } l_{ij} \text{ exists} \\ 1 & \text{otherwise} \end{cases} \end{aligned}$$

Because $\forall l^x_{ij} : (x > i) \wedge (x > j)$, and since activation values are computed top-down, n^x is computed before n^i, so l^x_{ij} does not receive input directly or indirectly from unit i. Its value is therefore not in any way a function of w_{ij}.

$$\frac{\partial n^i}{\partial w_{ij}} = s^j \tag{D.12}$$

$$\Delta w_{ij} = \frac{\partial E}{\partial w_{ij}} = \delta^i s^j \tag{D.13}$$

$$\delta^i = \begin{cases} \frac{\partial E}{\partial a^i} & \text{if } n^i \text{ is an action unit, } a^i \\ \frac{\partial E}{\partial l^i_{xy}} & \text{if } n^i \text{ is a higher-level unit, } l^i_{xy} \end{cases} \tag{D.14}$$

$$\frac{\partial E}{\partial a^i} = a^i - T^i \tag{D.15}$$

$$\begin{aligned} \frac{\partial E}{\partial l^i_{xy}} &= \frac{\partial E}{\partial \hat{w}_{xy}}\frac{\partial \hat{w}_{xy}}{\partial l^i_{xy}} \\ &= \frac{\partial E}{\partial \hat{w}_{xy}} \\ &= \frac{\partial E}{\partial n^x}\frac{\partial n^x}{\partial \hat{w}_{xy}} \\ &= \delta^x s^y \\ &= \Delta w_{xy}. \end{aligned} \tag{D.16}$$

Returning now to Equation D.13, and substituting in the results from Equations D.15 and D.16, the change to the weight can be written as:

$$\Delta w_{ij} = \delta^i s^j$$

$$= \quad s^j \begin{cases} a^i - T^i & \text{if } n^i \text{ is an action unit, } a^i \\ \Delta w_{xy} & \text{if } n^i \text{ is a higher-level unit, } l^i_{xy}. \end{cases} \tag{D.17}$$

If a fixed number of training patterns is given in advance, adding new units is also slightly different. In particular, long-term averages are not needed since exact averages can be computed.

Appendix E

Derivation of Learning Rule for Recurrent Network

The derivation of the learning rule for transition hierarchies with single recurrent self-connections is somewhat different from that of the non-recurrent system. Because the units have self-connections and can retain information without explicit use of time-delays, unit activations can be computed top-down as with the stationary network (Appendix D). Highest high-level units are computed first, down through lower-level l units, terminating with the action units. This means that all weights contribute to the network output at every time step. As in Appendix D, the equations here are (at least at first) labeled to correspond with those in Section 6.2.

Because of the self-connections, the input to the high-level and action units is:

$$n^i(t) = \hat{w}_{ii}(t)n^i(t-1) + \sum_j \hat{w}_{ij}(t)s^j(t-1) \tag{E.3}$$

For notational convenience, the current value of sensory unit j is now written $s^j(t-1)$ instead of $s^j(t)$. The definition of the higher-order connections is unchanged:

$$\hat{w}_{ij}(t) = \begin{cases} w_{ij} + l_{ij}(t-1) & \text{if a high-level unit } l_{ij} \text{ for weight } w_{ij} \text{ exists} \\ w_{ij} & \text{otherwise} \end{cases} \tag{E.4}$$

As before, the weights are changed in the direction opposite their contribution to the error, $E(t)$. Because all weights contribute to the output at every time step, the w_{ij}'s no longer need to be time-indexed. For the purposes of the derivation, the weights are assumed to remain the same at all time steps, though gradient-descent can still be approximated by making small weight-changes at every step.

$$\Delta w_{ij}(t) \stackrel{def}{=} \frac{\partial E(t)}{\partial w_{ij}}, \tag{E.6}$$

A weight w_{ij} is said to *affect* a unit u^k if w_{ij} either directly feeds into u^k (i.e., $k = i$) or w_{ij} feeds into a high-level unit l^i_{xy}, where w_{xy} affects u^k. Because each non-input unit has a self-connection, when a weight affects a unit at one time step, that effect influences the unit in the following time steps as well. Because of this, the derivation diverges at this point from that of Section 6.2.3:

$$\frac{\partial E(t)}{\partial w_{ij}} = \sum_k \frac{\partial E(t)}{\partial a^k(t)} \frac{\partial a^k(t)}{\partial w_{ij}}.$$

Each weight affects only one action unit, so

$$\frac{\partial E(t)}{\partial w_{ij}} = \frac{\partial E(t)}{\partial a^k(t)} \frac{\partial a^k(t)}{\partial w_{ij}}, \tag{E.7}$$

where a^k is the action unit that w_{ij} affects.

A new value ϕ_{ij}^k is enlisted into the derivation to hold the running contribution of weight w_{ij} to unit n^k. From Equation E.3:

$$\begin{aligned}\phi_{ij}^k(t) &\stackrel{def}{=} \frac{\partial n^k(t)}{\partial w_{ij}} \\ &= \frac{\partial \hat{w}_{kk}(t)}{\partial w_{ij}} n^k(t-1) + \frac{\partial n^k(t-1)}{\partial w_{ij}} \hat{w}_{kk}(t) + \sum_{j'} \frac{\partial \hat{w}_{kj'}(t)}{\partial w_{ij}} s^{j'}(t-1).\end{aligned}$$

If $k = i$, then w_{ij} feeds directly into u^k, so

$$\begin{aligned}\phi_{kj}^k(t) &= \begin{cases} n^k(t-1) + \phi_{kk}^k(t-1)\hat{w}_{kk}(t) + 0 & \text{if } k = j \\ 0 + \phi_{kj}^k(t-1)\hat{w}_{kk}(t) + s^j(t-1) & \text{if } k \neq j \end{cases} \\ &= \phi_{kj}^k(t-1)\hat{w}_{kk}(t) + u^j(t-1). \end{aligned} \tag{E.8}$$

If $k \neq i$, however, then w_{ij} affects a high-level unit (say l_{km}^x) that modifies weight w_{km} into unit k, then in this case:

$$\begin{aligned}\phi_{ij}^k(t) &= \begin{cases} \frac{\partial l_{kk}^x(t)}{\partial w_{ij}} n^k(t-1) + \phi_{ij}^k(t-1)\hat{w}_{kk}(t) + 0 & \text{if } k = m \\ 0 + \phi_{ij}^k(t-1)\hat{w}_{kk}(t) + \frac{\partial l_{km}^x(t)}{\partial w_{ij}} s^m(t-1) & \text{if } k \neq m \end{cases} \\ &= \phi_{ij}^k(t-1)\hat{w}_{kk}(t) + \phi_{ij}^x(t)u^m(t-1), \text{where } w_{ij} \text{ affects } l_{km}^x. \end{aligned} \tag{E.9}$$

Therefore, combining Equations E.8 and E.9,

$$\phi_{ij}^k(t) = \phi_{kj}^k(t-1)\hat{w}_{kk}(t) + \begin{cases} u^j(t-1) & \text{if } k = i \\ u^m(t-1)\phi_{ij}^x(t) & \text{Otherwise,} \\ & \text{where } w_{ij} \text{ affects } l_{km}^x. \end{cases} \tag{E.10}$$

From Equations E.6, E.7, and E.10:

$$\begin{aligned}\Delta w_{ij}(t) &= \frac{\partial E(t)}{\partial a^k(t)} \frac{\partial a^k(t)}{\partial w_{ij}} \\ &= (a^k(t) - T^k(t))\phi_{ij}^k(t),\end{aligned}$$

where a^k is the only action unit affected by w_{ij}, and the variable $\phi_{ij}^k(t)$ holds the derivative of unit k with respect to weight w_{ij} for all patterns seen up to time t.

The ϕ_{ij}^k values must be maintained and calculated at every time step. They are similar to the p_{ij}^k values of the RTRL algorithm as presented by Williams and Zipser [123]; though not so many of these values are required, since the network is not fully recurrent. However, it is because of this extra overhead that the recurrent variation of the algorithm lacks the elegance of the Temporal Transition Hierarchies network. For each weight, w_{ij}, the number of ϕ_{ij}^k values that must be stored is equal to the height of w_{ij} in the hierarchy (the same as τ^i in Section 6.2.3 for weight w_{ij}). The number of ϕ_{ij}^k values that must be stored therefore

scales with HN_w, where H is the height of the highest unit in the hierarchy and N_w is the number of network weights. The total number of operations required per network cycle also scales with HN_w. While this is better than the scaling behavior of the RTRL algorithm, it is not as good as Bachrach's, Mozer's, or Fahlman's algorithms [4, 27, 65] (Section 4.3).

One advantage of the recurrent version of transition hierarchies is that it does not require a record of previous activation values. (In the non-recurrent version such a record is required, and its length grows with the height of the hierarchy.) All values needed to compute the weight changes in this version are target or activation values from the current time step, or they are values held over from the previous time step only.

Bibliography

[1] J. S. Albus. Mechanisms of planning and problem solving in the brain. *Mathematical Biosciences*, 45:247–293, 1979.

[2] Ethem Alpaydin. GAL: Networks that grow when they learn and shrink when they forget. Technical Report 91–032, International Computer Science Institute, Berkeley, California, May 1991.

[3] C. W. Anderson. *Learning and Problem Solving with Multilayer Connectionist Systems*. PhD thesis, Department of Computer and Information Sciences, University of Massachusetts, 1986.

[4] Jonathan Richard Bachrach. Learning to represent state. Master's thesis, Department of Computer and Information Sciences, University of Massachusetts, Amherst, MA 01003, November 1988.

[5] Jonathan Richard Bachrach. *Connectionist Modeling and Control of Finite State Environments*. PhD thesis, Department of Computer and Information Sciences, University of Massachusetts, February 1992.

[6] L.C. Baird, III. Advantage updating. Technical report, Wright Laboratory, Wright–Patterson Air Force Base, OH, November 1993.

[7] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*, volume 1. William Kaufmann, Inc., Los Altos, California, 1981.

[8] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuron-like elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846, 1983.

[9] A. G. Barto, R. S. Sutton, and Christopher J. C. H. Watkins. Learning and sequential decision making. Technical Report COINS Technical Report 89–95, University of Massachusetts at Amherst, Department of Computer and Information Science, 1989.

[10] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Real-time learning and control using asynchronous dynamic programming. Technical Report 91–57, Computer Science Department, University of Massachusetts at Amherst, August 1991.

[11] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. Submitted to AI Journal special issue on Computational Theories of Interaction and Agency, January 1993.

[12] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-hall, Englewood Cliffs, NJ, 1989. Secondary source, from [11].

[13] Ulrich Bodenhausen and Alex Waibel. The Tempo 2 algorithm: adjusting time-delays by supervised learning. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 155–161, San Mateo, California, 1991. Morgan Kaufmann Publishers.

[14] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[15] D. Chen, C. L. Giles, G.Z. Sun, H. H. Chen, Y. C. Lee, and M. W. Goudreau. Constructive learning of recurrent neural networks. In *IEEE Proceedings of the ICNN*, 1993.

[16] Lonnie Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-92)*, Cambridge, MA, 1992. AAAI/MIT Press.

[17] Axel Cleeremans, David Servan-Schreiber, and James L. McClelland. Finite state automata and simple recurrent networks. *Neural Computation*, 1(3):372–381, 1989.

[18] Richard Dawkins. Hierarchical organisation: a candidate principle for ethology. In P. P. G. Bateson and R. A. Hinde, editors, *Growing Points in Ethology*, pages 7–54, Cambridge, 1976. Cambridge University Press.

[19] Shawn P. Day and Michael R. Davenport. Continuous-time temporal back-propagation with adaptable time delays. Submitted to: IEEE Transactions on Neural Networks, August 1991.

[20] Peter Dayan, May 1994. Personal Communication.

[21] Peter Dayan and Geoffrey E. Hinton. Feudal reinforcement learning. In C. L. Giles, S. J. Hanson, and J. D. Cowan, editors, *Advances in Neural Information Processing Systems 5*, pages 271–278, San Mateo, California, 1993. Morgan Kaufmann Publishers.

[22] Peter Dayan and Terrence J. Sejnowski. TD(λ) converges with probability 1. Submitted to Machine Learning, 1993.

[23] Kenji Doya. Universality of fully-connected recurrent neural networks. Submitted to IEEE Transactions on Neural Networks, 1993.

[24] Gary L. Drescher. *Made-Up Minds: A Constructivist Approach to Artificial Intelligence*. MIT Press, Cambridge, Massachusetts, 1991.

[25] Jeffrey L. Elman. Finding structure in time. CRL Technical Report 8801, University of California, San Diego, Center for Research in Language, April 1988.

[26] Scott E. Fahlman. Faster learning variations on back-propagation: An empirical study. Technical Report CMU–CS–88–162, Carnegie Mellon University, School of Computer Science, September 1988.

[27] Scott E. Fahlman. The recurrent cascade-correlation architecture. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 190–196, San Mateo, California, 1991. Morgan Kaufmann Publishers.

[28] Scott E. Fahlman and Christian Lebiere. The Cascade-Correlation learning architecture. Technical Report CMU–CS–90–100, Carnegie Mellon University, School of Computer Science, February 1990.

[29] Gerald Fahner. A higher order unit that performs arbitrary boolean functions. In *Proceedings of the International Joint Conference on Neural Networks*, volume III, pages 193–197, 1990.

[30] Gerald Fahner and Rolf Eckmiller. Structural adaptation of parsimonious higher order neural classifiers. Technical report, Department of Neuroinformatics, University of Bonn, Bonn, Germany, 1992.

[31] Marcus Frean. The Upstart algorithm: a method for constructing and training feed-forward neural networks. Preprint 89/469, Department of Physics and Centre for Cognitive Science, Edinburgh University, 1989.

[32] Stephen I. Gallant. Three constructive algorithms for network learning. In *The Eighth Annual Conference of the Cognitive Science Society*, pages 652–660. Lawrence Erlbaum, August 1986.

[33] C. L. Giles, C. B. Miller, D. Chen, G. Z. Sun, H. H. Chen, and Y. C. Lee. Extracting and learning an unknown grammar with recurrent neural networks. In J. E. Moody, S. J. Hanson, and R. P. Lippman, editors, *Advances in Neural Information Processing Systems 4*, pages 317–324, San Mateo, California, 1992. Morgan Kaufmann Publishers.

[34] C. L. Giles, G. Z. Sun, H. H. Chen, Y. C. Lee, and D. Chen. Higher order recurrent networks & grammatical inference. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, San Mateo, California, 1990. Morgan Kaufmann Publishers.

[35] C. Lee Giles and Tom Maxwell. Learning, invariance, and generalization in high-order neural networks. *Applied Optics*, 26(23):4972–4978, December 1987.

[36] Mark W. Goudreau, C. Lee Giles, Srimat T. Chakradhar, and D. Chen. First-order vs. second-order single layer recurrent neural networks. *IEEE Transactions on Neural Networks*, 1993.

[37] G. R. Grimmett and D. R. Stirzaker. *Probability and Random Processes.* Oxford University Press, New York, 1985.

[38] Stephen Grossberg. A theory of human memory: Self-organization and performance of sensory-motor codes, maps, and plans. In R. Rosen and F. Snell, editors, *Progress in Theoretical Biology, Vol. 5*, pages 223–374. Academic Press, New York, 1978.

[39] J. H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, Michigan, 1975.

[40] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[41] Jenq-Neng Hwang and Chi H. Chan. Iterative constrained inversion of neural networks and its applications. In *The 24th Conference on Information Systems and Sciences*, pages 754–759, Princeton, March 1990.

[42] John W. Jameson. Reinforcement control with hierarchical backpropagated adaptive critics. Submitted to *Neural Networks*, March 1992.

[43] Michael I. Jordan. Serial order: A parallel distributed processing approach. ICS Report 8604, Institute for Cognitive Science, University of California, San Diego, May 1986.

[44] Michael I. Jordan. Generic constraints on underspecified target trajectories. In *Proceedings of the International Joint Conference on Neural Networks*, volume I, pages 217–225, 1989.

[45] Michael I. Jordan and Robert A. Jacobs. Learning to control an unstable system with forward modeling. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 324–331, San Mateo, California, 1990. Morgan Kaufmann Publishers.

[46] Michael I. Jordan and David E. Rumelhart. Forward models: Supervised learning with a distal teacher. *Cognitive Science*, 16:307–354, 1992.

[47] Leslie Pack Kaelbling. Hierarchical learning in stochastic domains: Preliminary results. In *Machine Learning: Proceedings of the tenth International Conference*, pages 167–173. Morgan Kaufmann Publishers, June 1993.

[48] Leslie Pack Kaelbling. Learning to achieve goals. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1094–1098, Chambéry, France, 1993. Morgan Kaufmann.

[49] J. Kindermann and A. Linden. Inversion of neural networks by gradient descent. *Journal of Parallel Computing*, 14(3), 1990.

[50] A. H. Klopf. *The Hedonistic Neuron: A Theory of Memory, Learning, and Intelligence*. Hemisphere, Washington, 1982.

[51] Benjamin J. Kuipers and Yung-Tai Byun. A robust, qualitative method for robot spatial learning. In *The Seventh National Conference on Artificial Intelligence*, 1988.

[52] John E. Laird, Paul S. Rosenbloom, and Alan Newell. Chunking in soar: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[53] Thibault Langlois and Stéphane Canu. Control of time-delay systems using reinforcement learning. In Igor Alexsander and John Taylor, editors, *Artificial Neural Networks, 2: Proceedings of the 1992 International conference on Artificial Neural Networks (ICANN-92)*, pages 607–610, Amsterdam, 1992. North-Holland, Elsevier Science Publishing.

[54] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

[55] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, Carnegie Mellon University, 1993. Also appears as Technical Report CMU-CS-93-103.

[56] Alexander Linden. On discontinuous Q-functions in reinforcement learning. In H. J. Ohlbach, editor, *Proceedings of the German Workshop on Artificial Intelligence.* Springer Verlag, 1993.

[57] Alexander Linden and Frank Weber. Implementing inner drive through competence reflection. In J. A. Meyer, H. Roitblat, and S. Wilson, editors, *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 321–326. MIT Press, 1993.

[58] Michael L. Littman. An optimization-based categorization of reinforcement learning environments. In J. A. Meyer, H. Roitblat, and S. W. Wilson, editors, *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 262–270. MIT Press, 1993.

[59] R. Andrew McCallum. Overcoming incomplete perception with Utile Distinction Memory. In *Machine Learning: Proceedings of the Tenth International Conference*, pages 190–196. Morgan Kaufmann Publishers, June 1993.

[60] Clifford B. Miller and C.L. Giles. Experimental comparison of the effect of order in recurrent neural networks. *International Journal of Pattern Recognition and Aritificial Intelligence*, 1993. Special Issue on Applications of Neural Networks to Pattern Recognition.

[61] Marvin Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.

[62] Marvin L. Minsky and Seymour A. Papert. *Perceptrons.* MIT Press, Cambridge, MA, 1969.

[63] John Moody and Christian Darken. Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1(2):281–294, 1989.

[64] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13(1), 1993.

[65] Michael C. Mozer. A focused back-propagation algorithm for temporal pattern recognition. Technical Report CRG–TR–88–3, Department of Psychology, University of Toronto, June 1988.

[66] Michael C. Mozer. Induction of multiscale temporal structure. In John E. Moody, Steven J. Hanson, and Richard P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 275–282, San Mateo, California, 1992. Morgan Kaufmann Publishers.

[67] Michael C. Mozer and Sreerupa Das. A connectionist symbol manipulator that discovers the structure of context-free languages. In C. L. Giles, S. J. Hanson, and J. D. Cowan, editors, *Advances in Neural Information Processing Systems 5*, pages 863–870, San Mateo, California, 1993. Morgan Kaufmann Publishers.

[68] Paul Munro. A dual back-propagation scheme for scalar reward learning. In *The Ninth Annual Conference of the Cognitive Science Society*, pages 165–176, Hillsdale, NJ, 1987. Lawrence Erlbaum.

[69] Derrick Nguyen and Bernard Widrow. The truck backer-upper: An example of self-learning in neural networks. In W. Thomas Miller, III, Richard S. Sutton, and Paul J. Werbos, editors, *Neural Networks for Control*, chapter 12, pages 288–299. MIT Press, 1990.

[70] Athanasios Papoulis. *Probability, Random Variables, and Stochastic Processes.* McGraw-Hill Series in Systems Science. McGraw-Hill, 1965.

[71] Jing Peng and Ronald J. Williams. Efficient learning and planning within the Dyna framework. In J. A. Meyer, H. Roitblat, and S. Wilson, editors, *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 281–290. MIT Press, 1993.

[72] David Pierce and Benjamin Kuipers. Learning hill-climbing functions as a strategy for generating behaviors in a mobile robot. In J. A. Meyer and S. W. Wilson, editors, *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 327–336. MIT Press, 1991.

[73] David Pierce and Benjamin Kuipers. Learning to explore and build maps. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-94)*, Cambridge, MA, 1994. AAAI/MIT Press. To appear.

[74] Jordan B. Pollack. Cascaded back-propagation on dynamic connectionist networks. In *The Ninth Annual Conference of the Cognitive Science Society*, pages 391–404, 1987.

[75] Jordan B. Pollack. *On Connectionist Models of Natural Language Processing.* PhD thesis, Computer Science Department, University of Illinois, Urbana, IL, 1987.

[76] Jordan B. Pollack. The induction of dynamical recognizers. *Machine Learning*, 7:227–252, 1991.

[77] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 1992.

[78] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77, 1989.

[79] A. S. Reber. Implicit learning of artificial grammars. *Journal of Verbal Learning and Verbal Behavior*, 5:855–863, 1967. Secondary source, from [17].

[80] Mark B. Ring. Incremental development of complex behaviors through automatic construction of sensory-motor hierarchies. In Lawrence A. Birnbaum and Gregg C. Collins, editors, *Machine Learning: Proceedings of the Eighth International Workshop (ML91)*, pages 343–347. Morgan Kaufmann Publishers, June 1991.

[81] Mark B. Ring. Learning sequential tasks by incrementally adding higher orders. In C. L. Giles, S. J. Hanson, and J. D. Cowan, editors, *Advances in Neural Information Processing Systems 5*, pages 115–122, San Mateo, California, 1993. Morgan Kaufmann Publishers.

[82] Mark B. Ring. Sequence learning with incremental higher-order neural networks. Technical Report AI 93–193, Artificial Intelligence Laboratory, University of Texas at Austin, January 1993.

[83] Mark B. Ring. Two methods for hierarchy learning in reinforcement environments. In J. A. Meyer, H. Roitblat, and S. Wilson, editors, *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 148–155. MIT Press, 1993.

[84] R. L. Rivest and R. E. Schapire. A new approach to unsupervised learning in deterministic environments. In Pat Langley, editor, *Proceedings of the Fourth International Workshop on Machine Learning*, pages 364–375, Irvine, CA, June 1987.

[85] A. J. Robinson and F. Fallside. The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department, 1987.

[86] H. L. Roitblat. A cognitive action theory of learning. In J. Delacour and J. C. S Levy, editors, *Systems with Learning and Memory Abilities*, pages 13–26. Elsevier Science Publishers B.V. (North-Holland), 1988.

[87] H. L. Roitblat. Cognitive action theory as a control architecture. In J. A. Meyer and S. W. Wilson, editors, *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 444–450. MIT Press, 1991.

[88] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. V1: Foundations*. MIT Press, 1986.

[89] Terence D. Sanger. A tree structured adaptive network for function approximation in high-dimensional spaces. *IEEE Transactions on Neural Networks*, 2(2):285–301, March 1991.

[90] Terence D. Sanger, Richard S. Sutton, and Christopher J. Matheus. Iterative construction of sparse polynomial approximations. In J. E. Moody, S. J. Hanson, and R. P. Lippman, editors, *Advances in Neural Information Processing Systems 4*, pages 1064–1071, San Mateo, California, 1992. Morgan Kaufmann Publishers.

[91] Jürgen Schmidhuber. Making the world differentiable: On using self-supervised fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environments. Technical Report FKI–126–90 (revised), Technische Universität München, Institut für Informatik, November 1990.

[92] Jürgen Schmidhuber. Networks adjusting networks. Technical Report FKI–125–90 (revised), Technische Universität München, Institut für Informatik, November 1990.

[93] Jürgen Schmidhuber. Adaptive confidence and adaptive curiosity. Technical Report FKI–149–91 (revised), Technische Universität München, Institut für Informatik, April 1991.

[94] Jürgen Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In J. A. Meyer and S. W. Wilson, editors, *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 15–21. MIT Press, 1991.

[95] Jürgen Schmidhuber. A fixed size storage $O(n^3)$ time complexity learning algorithm for fully recurrent continually running networks. *Neural Computation*, 4(2):243–248, 1992.

[96] Jürgen Schmidhuber. Learning unambiguous reduced sequence descriptions. In J. E. Moody, S. J. Hanson, and R. P. Lippman, editors, *Advances in Neural Information Processing Systems 4*, pages 291–298, San Mateo, California, 1992. Morgan Kaufmann Publishers.

[97] Jürgen Schmidhuber and Reiner Wahnsiedler. Planning simple trajectories using neural subgoal generators. In J. A. Meyer, H. Roitblat, and S. Wilson, editors, *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 196–199. MIT Press, 1993.

[98] Terrence J. Sejnowski and Charles R. Rosenberg. Parallel networks that learn to pronounce english text. *Complex Systems*, 1:145–168, 1987.

[99] Satinder Pal Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8, May 1992.

[100] Satinder Pal Singh, Andrew G. Barto, Roderic Grupen, and Christopher Connolly. Robust reinforcement learning in motion planning. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 655–662, San Mateo, California, 1994. Morgan Kaufmann Publishers.

[101] G. Z. Sun, H. H. Chen, C. L. Giles, Y. C. Lee, and D. Chen. Connectionist push-down automata that learn context-free grammars. In *Proceedings of the International Joint Conference on Neural Networks*, pages I–577–580, Hillsdale, NJ, 1990. Erlbaum Associates.

[102] Guo-Zhen Sun, Hsing-Hen Chen, and Lee Yee-Chun. Green's function method for fast on-line learning algorithm of recurrent neural networks. In J. E. Moody, S. J. Hanson, and R. P. Lippman, editors, *Advances in Neural Information Processing Systems 4*, pages 333–340, San Mateo, California, 1992. Morgan Kaufmann Publishers.

[103] Richard S. Sutton. *Temporal Credit Assignment in Reinforcement Learning.* PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003, 1984.

[104] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[105] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In Bruce W. Porter and Ray J. Mooney, editors, *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224. Morgan Kaufmann Publishers, June 1990.

[106] Richard S. Sutton. Reinforcement learning architectures for animats. In J. A. Meyer and S. W. Wilson, editors, *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 288–296. MIT Press, 1991.

[107] Sebastian B. Thrun and Knut Möller. Active exploration in dynamic environments. In J. E. Moody, S. J. Hanson, and R. P. Lippman, editors, *Advances in Neural Information Processing Systems 4*, pages 531–538, San Mateo, California, 1992. Morgan Kaufmann Publishers.

[108] Sebastian B. Thrun, Knut Möller, and Alexander Linden. Planning with an adaptive world model. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, San Mateo, California, 1991. Morgan Kaufmann Publishers.

[109] M. Tomita. Dynamic construction of finite-state automata from examples using hill-climbing. In *Proceedings of the Fourth Annual Cognitive Science Conference*, pages 150–108, Ann Arbor, MI, 1982. Secondary source, from [76].

[110] Alex Waibel. Modular construction of time-delay neural networks for speech recognition. *Neural Computation*, 1(1):39–46, Spring 1989.

[111] Christopher J. C. H Watkins. *Learning from Delayed Rewards.* PhD thesis, King's College, May 1989.

[112] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[113] Raymond L. Watrous and Gary M. Kuhn. Induction of finite-state languages using second-order recurrent networks. In J. E. Moody, S. J. Hanson, and R. P. Lippman, editors, *Advances in Neural Information Processing Systems 4*, pages 309–316, San Mateo, California, 1992. Morgan Kaufmann Publishers.

[114] Paul J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1:339–356, 1988.

[115] Paul J. Werbos. Backpropagation through time: What it is and how to do it. *Proceedings of the IEEE*, October 1990.

[116] Paul J. Werbos. A menu of designs for reinforcement learning over time. In W. Thomas Miller, III, Richard S. Sutton, and Paul J. Werbos, editors, *Neural Networks for Control*, chapter 3, pages 67–95. MIT Press, 1990.

[117] Steven D. Whitehead and Dana H. Ballard. Active perception and reinforcement learning. *Neural Computation*, 2(4):409–419, 1990.

[118] B. Widrow and M. E. Hoff. Adaptive switching circuits. In *Institute of Radio Engineers, Western Electronic Show and Convention, Convention Record*, pages 96–104, New York, 1960.

[119] Ronald J. Williams. Inverting a connectionist network mapping by back-propagation of error. In *The Eighth Annual Conference of the Cognitive Science Society*, pages 859–865. Lawrence Erlbaum, August 1986.

[120] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[121] Ronald J. Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 2(4):490–501, 1990.

[122] Ronald J. Williams and David Zipser. Experimental analysis of the real-time recurrent learning algorithm. *Connection Science*, 1(1):87–111, 1989.

[123] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.

[124] S. W. Wilson. Hierarchical credit allocation in a classifier system. In M. S. Elzas, T. I. Ören, and B. P. Zeigler, editors, *Modeling and Simulation Methodology*. Elsevier Science Publishers B.V., 1989.

[125] Stewart W. Wilson. The animat path to AI. In J. A. Meyer and S. W. Wilson, editors, *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 15–21. MIT Press, 1991.

[126] Lambert E. Wixson. Scaling reinforcement learning techniques via modularity. In Lawrence A. Birnbaum and Gregg C. Collins, editors, *Machine Learning: Proceedings of the Eighth International Workshop (ML91)*, pages 368–372. Morgan Kaufmann Publishers, June 1991.

[127] Mike Wynn-Jones. Node splitting: A constructive algorithm for feed-forward neural networks. *Neural Computing and Applications*, 1(1):17–22, 1993.