University of Alberta

Temporal Abstraction in Temporal-difference Networks

by

Eddie JR Rafols

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2006

Abstract

The quality of a knowledge representation directly influences an agent's ability to interact with an environment. Temporal-difference (TD) networks, a recently introduced knowledge representation framework, model a world with a set of action-conditional predictions about sensations. Some key characteristics of TD networks are that they: 1) relate knowledge to sensations, 2) allow the agent to make predictions about other predictions (compositionality) and 3) provide a means for abstraction. The focus of this thesis is connecting high-level concepts to data by abstracting over space and time. Spatial abstraction in TD networks helps with scaling issues by grouping situations with similar sets of predictions into abstract states. A set of experiments demonstrate the advantages of using the abstract states as a representation for reinforcement learning. Temporal abstraction is added to TD networks by extending the framework to predict arbitrarily distant future outcomes. This extension is based on the options framework, an approach to including temporal abstraction in reinforcement-learning algorithms. Including options in the TD-network framework brings about a challenging problem: learning about multiple options from a single stream of data (also known as off-policy learning). The first algorithm for the off-policy learning of predictions about option outcomes is introduced in this thesis.

Acknowledgements

While only my name appears on the cover of this thesis, I could not have written it without the help of many others. Thanks to Dr. Rich Sutton, my supervisor, who introduced me to the research found in this thesis and who gave me a new perspective on the idea of artificial intelligence. I would also like to thank Dr. Michael Bowling and Dr. Petr Musilek for participating on my thesis committee. Thanks go to Dr. Mark Ring, who took the time to thoroughly proofread the initial draft of this thesis. Many thanks go to Brian Tanner, Anna Koop, Cosmin Paduraru and the rest of the RLAI group who provided help and friendship over the past two years (and who also managed to tolerate me for two years). And finally, I would like to thank my parents, Eddie and Evangeline Rafols, for all their encouragement and support. This thesis is a reflection of the love for learning you instilled in me.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Knowledge representation is a critical issue in the field of artificial intelligence. An artificial-intelligence agent interacts with an environment. The agent's understanding of the dynamics of the world is summarized by its knowledge representation. If the agent is charged with completing a task in this world, the quality of the knowledge representation can make the difference between the agent's success and failure. In this thesis I address a major challenge of knowledge representation: forming high-level concepts from low-level observations. Just as a person can form an understanding of the world from their nerve impulses, a learning agent will, ideally, form a representation of the environment from its own sensations. The main contribution of this thesis is an approach to knowledge representation that attempts to bridge the gap between low-level observations and high-level concepts.

1.1 Predictive Representations

Predictive representations are a recent development in knowledge representation that connect knowledge to *experience*—in this thesis, a sequence of action-observation pairs. Actions, chosen according to some behavior policy, are taken by the agent, and observations are emitted by the environment in response.

Predictive representations encapsulate knowledge as predictions about future experience. Correct predictions about the outcomes of possible interactions with the environment demonstrate an understanding of the environment. For example, a basketball can be manipulated in many different ways and there is a corresponding prediction about the outcome of each manipulation.

- If I rotate the ball, I expect to observe a certain pattern on the other side of the ball.
- If I bounce the ball, I expect to observe the ball following a certain trajectory.

- If I pick up the ball, I expect to observe the ball in my hands (the observation in this case may be tactile rather than visual as in the previous two manipulations).

Being able to predict what I will observe given many different interactions is a form of knowledge about the basketball. In turn, being able to distinguish between an inflated basketball and a flat basketball involves knowing that the same manipulations (rotating, bouncing, picking up, etc.) result in different observations. While the observation for the two balls may be similar, a dramatically different observation is expected if the balls are bounced.

An important characteristic of predictive representations is that the representation is *subjective* to the agent—knowledge is represented with respect to the agent's experience. This approach to knowledge representation is a departure from knowledge representation as in expert systems, where knowledge is a set of arbitrary symbols. Learning or verifying these symbols requires an oracle that can interpret and provide meaning to the symbols. In contrast, knowledge in a predictive representation is both learnable and verifiable by the agent because knowledge is represented as quantities that the agent can observe and actions that the agent can take.

Another important characteristic of predictive representations is that predictions can be used as state—the current predictions are computed from the previous set of predictions, and the next set of predictions are computed from the current set of predictions. State, in a predictive representation, is therefore internal to the agent. This differs from many other representations in which state is a property of the environment and is not always observable by the agent. Predictive representations are particularly useful in the absence of an observable environmental state because rather than attempting to reconstruct the latent environmental state (which is tough to accomplish from data), the agent can use its actions and the available observations to represent the environment.

1.2 Temporal-difference Networks

Temporal-difference networks, recently introduced by Sutton and Tanner (2004), represent knowledge as a set of predictions about future interactions with the world and are thus predictive representations. The distinguishing feature of TD networks is that they permit a compositional specification of the quantity being predicted—predictions are made not only about specific observable quantities, but also about other predictions.

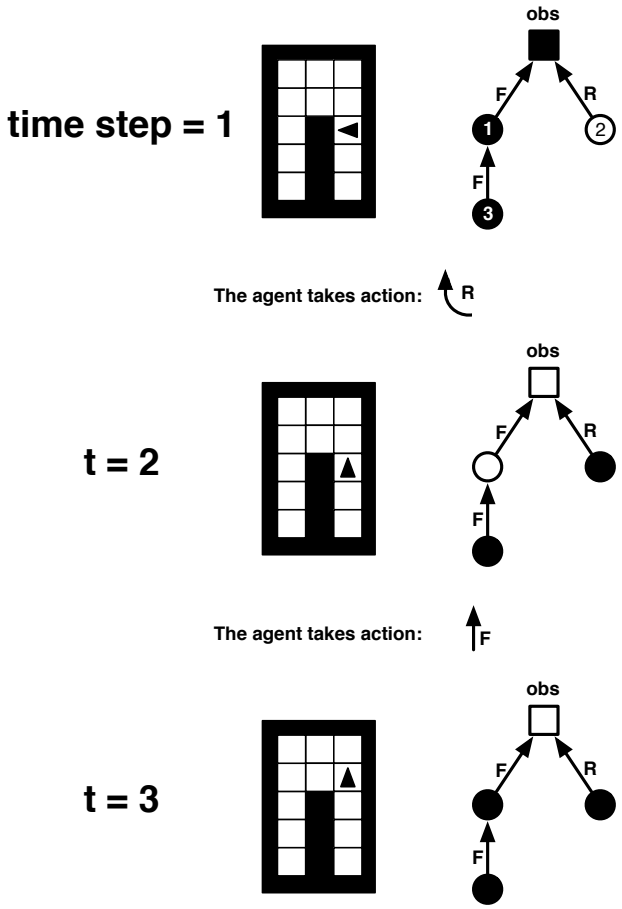


Figure 1.1: This example illustrates an agent in a grid-world that rotates (R) and then steps forward (F). The labels on the left indicate the time index. The agent's position is demonstrated in the second column, while the TD network pictured in the third column illustrates the predictions made on each time-step. The square represents the current observation; the square is black if the agent is facing a wall and the square is white if the agent is facing an empty cell. The nodes of the TD network (circles) represent predictions, and the arrows indicate the predictive target, conditional on the labeled action. The color of each circle indicates the correct prediction.

Compositionality can be described in the context of the basketball example. After each manipulation, new predictions can be made about future manipulations—depending on whether I pick up a ball with my right hand or my left hand, I may make two different sets of predictions about what I will observe if I then bounce the ball.

An example grid world problem and a corresponding temporal-difference network is pictured in Figure 1.1. The agent, represented by a triangle, can be in one of four orientations: North, South, East, or West. The triangle representing the agent is pointed in the direction that the agent is facing (e.g., at time step 1 the agent is facing West). On each time step the agent chooses one of the two actions: step forward (F) or Rotate (R). If the agent is

facing a wall (black grid cell), then the step forward action will have no effect; if the agent is facing an open space (white grid cell), then the step-forward action will advance the agent one grid cell in the direction it is facing. The rotate action causes the agent to rotate 90° clockwise while remaining in the same grid cell. The agent observes the color of the grid cell that it is facing. For example, at time step 1 the agent observes black; at time step 2 the agent observes white (the current observation is represented by the square marked **obs** at each time step of Figure 1.1).

An example TD network is pictured to the right of the grid world. The agent's current observation is represented by a square while its predictions are represented by circles. The arrows indicate the quantity being predicted (also called the *target* of prediction), while the label on the arrows indicate that the prediction is *action-conditional* (the agent predicts the value of the target *if* a certain action were taken). The targets have a temporal aspect—each node predicts the value of its target on the *next* time step.

The prediction of the node labeled 1 can be interpreted as asking the question: "What will the agent observe if it steps forward?" At time step 1, the circle for Node 1 is filled with black to indicate that the correct prediction is that the agent will observe a black cell. Similarly, Node 2 asks the question: "What will the agent observe if it rotates?" At time step 1, if the rotate action were taken then the agent would be facing a white grid cell so the circle for Node 2 is filled with white. Node 3 asks: "What will the value of node 1 be if the agent steps forward?" This question illustrates a compositional prediction (Node 3 is making a prediction about another prediction.) Node 3 asks a question about Node 1; Node 1 asks a question about the observation. Node 3 is therefore asking a question about the value of the observation two time steps in the future: "What will the agent observe if it steps forward, then steps forward again?" This *extensive* question being asked by node 3 is the question asked if the chain of compositions is followed from a node until an observation (the extensive question is thus *grounded* in the observation). Notice that each node in the network is framed as a *question* about future interactions. The three node and one observation structure in Figure 1.1 is referred to as the *question network* of this particular temporal-difference network.

The second row ($t = 2$) of Figure 1.1 illustrates how the predictions in the TD network change after the agent rotates. Stepping forward would result in an observation of a white grid cell; stepping forward twice would result in an observation of a black grid cell; rotating would result in an observation of a black grid cell as well. The change is reflected in the

question network by the color of the node. The third row ($t = 3$) shows the corresponding changes after the agent takes the step forward action.

This example only illustrates the question network, which defines the agent's predictions. There is also an underlying *answer network*, which specifies how predictions are updated on each time step. Chapter 4 provides a formal description of both the question network and the answer network.

1.3 Abstraction

Abstraction is the process of transforming a base set of objects into a more general set of *abstract* objects based on commonalities. A simple example of abstraction is the aggregation of physically proximal locations into a common group: rooms can be abstracted into houses, houses can be abstracted into neighborhoods, neighborhoods can be abstracted into cities.

Abstraction becomes increasingly important as the environments being modeled grow in size and complexity. In a large state space, it its often impractical to treat each state separately. State abstraction can produce a smaller, abstract state space that captures underlying regularities in the environment. Returning to the basketball example, a court can be abstracted into regions (offensive zone, defensive zone, within shooting range, etc.) rather than considering every single position on the court as a separate location. Also, in a large state space, the effect of a primitive action may be negligible, but extended sequences of actions may have perceivable effects in the world. Temporal abstraction can reduce long sequences of primitive actions into high-level units of action. In the basketball example, a sequence of low level actions can be abstracted into an extended way of behaving. Temporal abstraction allows shooting the ball to be modeled as a singular, temporally-extended unit of action rather than treating each muscle twitch in the shooting motion as a separate unit of action. Examples of both types of abstraction, spatial and temporal, are found in this thesis.

In an experience-oriented representation, there are often commonalities between sequences of (both past and future) experience. In a predictive representation, state can be abstracted by grouping situations with similar sets of predictions. Experiments conducted in Chapter 3 attempt to ascertain the quality of the generalization effected by predictive representations. These experiments use the predictions of a TD network to test the *predictive representations hypothesis* which holds that representing state as predictions is particularly

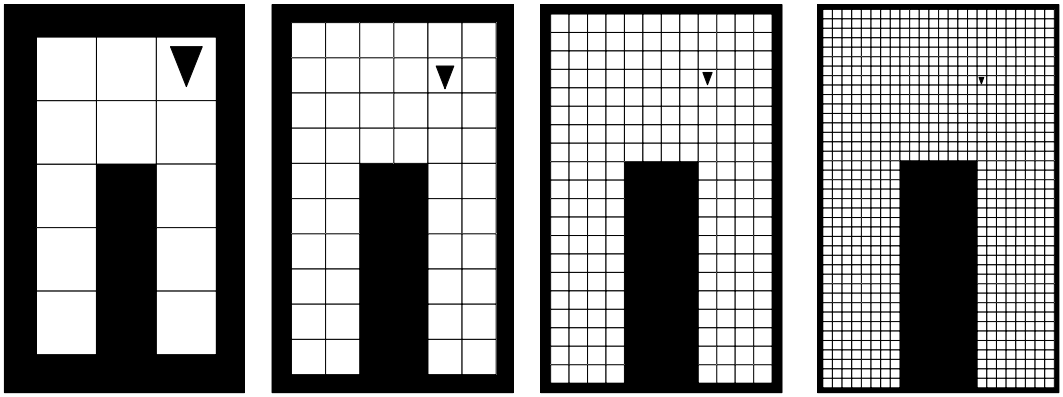


Figure 1.2: The example grid-world from Figure 1.1 grows larger as the granularity increases. The larger worlds can be modeled as predictions about temporally extended behaviors, such as "step forward until a wall is observed" or "with a %50 chance of terminating on each time-step, step forward until termination".

good for generalization. TD networks abstract over state, but in the existing framework they do not abstract over time.

Temporal abstraction, dealt with in detail in this thesis, can be carried out by treating action sequences of arbitrary lengths as singular units. Modeling small worlds at the lowest level of interaction (i.e., in terms of single-step actions) is feasible, but it quickly becomes impractical to model environments at this low level as they grow larger and more complex. For example, while simple actions may suffice to model the grid-world presented in Figure 1.1, suppose each grid-cell is split into four smaller cells. Now imagine that the granularity of this grid-world is continually increased until the agent is but a speck in a sea of white grid cells (as suggested by Figure 1.2). Modeling the environments in Figure 1.2 as single-step predictions requires a larger number of predictions each time the world increases in size (in each subsequent world, each step-forward action will have to be replaced by two steps forward). In contrast, the world can be modeled as a set of temporally abstract predictions about the outcome of temporally extended behaviors. Predictions could be made about observations arbitrarily distant in the future such as a prediction for the outcome of stepping forward until hitting a wall or a prediction for the probability of reaching a wall if always stepping forward, but with a %50 chance of the extended-action's termination on each step. These predictions can be made regardless of the size of the world and thus, despite the fact that the world is increasing in size, a fixed set of temporally extended predictions could capture the general structure of the environment.

The *options* framework (Sutton, Precup, & Singh, 1999) abstracts actions into tempo-

rally extended action-sequences. An option is defined by its three components:

- a set of situations from which the option can be initiated;
- a behavior policy, which determines how the agent is to act in any given situation;
- a set of situations in which the option may terminate.

The TD-network framework is extended to included options, forming a new temporally abstract modeling algorithm. Chapters 4,5, and 6 deal with combining options with temporal-difference networks and the associated problem of learning about multiple options from a single stream of data.

1.4 Temporal Abstraction in Temporal-difference Networks

This thesis extends the temporal-difference network framework to accommodate temporally abstract predictions, and it explores issues that arise when attempting to learn these long-term predictions. Section 1.2 provided an example of a temporal-difference network, in which the predictive targets were conditioned on actions. In the extended framework, targets are conditioned on options. Option-conditional predictions now ask questions of the general form: "What will the value of the target be if the agent executes the option until termination?"

Figure 1.3 suggests the increased representational power of an option-conditional TD (OTD) network. The network is now modeling a situation in the game of basketball. The observation is whether a basket is scored while the options are Dribble, Shoot, and Pass. Prediction 1 asks the question: "If I shoot the ball, will I observe a basket?", prediction 2 asks: "If I pass the ball, will I observe a basket?" Prediction 3 is a compositional prediction which asks the question: "If I dribble the ball up the court, what will the value of prediction 1 be?", or in extensive form: "If I dribble the ball up the court, then shoot the ball, will I observe a basket?" The question network in Figure 1.3 is structurally identical to the question network in Figure 1.1, but the predictions made in Figure 1.3 are now option-conditional.

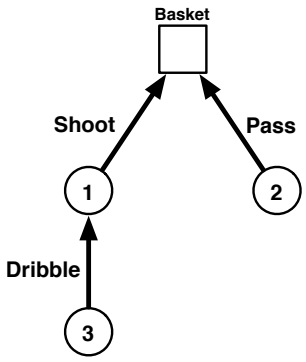


Figure 1.3: An example of a temporal-difference network with temporal abstraction. The network shares the same structure as Figure 1.1, but now the targets are conditioned on options rather than simple actions.

1.5 Off-policy Learning

An OTD network may consist of many predictions, each corresponding to an option. As the agent interacts with the environment, multiple option policies may be similar to the policy the agent uses to generate actions. Many option policies are similar to the behavior policy. An efficient use of data is to update all predictions associated with these options. Learning about one policy while following a different policy is known as off-policy learning. However, when combined with temporal-difference methods, off-policy learning may not converge to a solution—predictions may grow without bounds. Chapter 5 explores the general problem of off-policy learning and Chapter 6 studies the problem as it applies to temporal-difference networks.

1.6 Outline

This thesis progresses as follows. Chapter 2 is a survey of experience-oriented approaches to learning world models. The survey covers both the predecessors and the contemporaries of temporal-difference networks. Chapter 3 explores spatial abstraction and studies potential advantages of using predictions as state. Experimental results suggest that predictive representations usefully abstract over state because they generalize well. The work in Chapter 3 is independent from the following three chapters as it deals with *what can be represented*, whereas Chapters 4, 5, and 6 deal with another important issue in knowledge representation: *how a representation is learned*. Chapter 4 presents the first algorithm for the on-policy learning OTD networks. The new algorithm successfully learns a model of

a partially observable grid-world environment and the emergence of a learned concept is demonstrated. Chapter 5 addresses instability that may occur when off-policy learning is combined with function approximation and TD methods. A provably sound algorithm for the off-policy learning of option models is introduced in this chapter. Chapter 6 combines the research of the previous two chapters into an algorithm for the off-policy learning of OTD networks. Chapters 4, 5, and 6 are related in that the work in each subsequent chapter builds on the previous chapter's work and the experiments presented in these chapters were conducted on a common testbed. Chapter 7 summarizes the new algorithms presented in this thesis, suggests possible future avenues of research related to this thesis and discusses the implications of the experimental results these algorithms.

Chapter 2

Related Work

This chapter is a survey of work related to this thesis. Many algorithms, TD networks included, model a class of environments known as discrete dynamical systems. We first present a formal description of discrete dynamical systems and relate them to other systems. Next, we present a brief description of temporal-difference learning and the temporal-difference network framework. The last section of this chapter is a description of grounded representations—representations that are similar to temporal-difference networks in that they represent knowledge in terms of actions and observations.

2.1 Discrete Dynamical Systems

In this work, algorithms are developed to model discrete dynamical systems (DDS). In these systems, an agent interacts with an environment by taking actions and receiving observations (Figure 2.1). At discrete time step t the learning agent is in environmental state $s_t \in \mathcal{S}$ and selects an action, $a_t \in \mathcal{A}$. The action provokes a change in the environmental state from s_t to s_{t+1} according to probability $\mathcal{P}^{a_t}_{s_t s_{t+1}}$. As a result of the transition, the environment emits an observation $o_{t+1} \in \mathcal{O}$.

The term *experience* refers to a sequence of interactions between the agent and the environment in the form $a_i, o_i, a_{i+1}, o_{i+1}, \cdots, a_n, o_n$. History, h_t, is a specific stream of experience that spans from the beginning of time to the current time step: $a_0, o_0, a_1, o_1, \cdots, a_{t-1}$, o_{t-1}.

If $o_{t+1} = s_{t+1}$ then the agent observes the environmental state, or *Markov* state, and the observation summarizes the entire history. However, in the partially observable case o_{t+1} is a discrete symbol or set of symbols which do not uniquely identify the agent's current state; that is, the observation may be a single bit of information (as in Chapter 3) or the

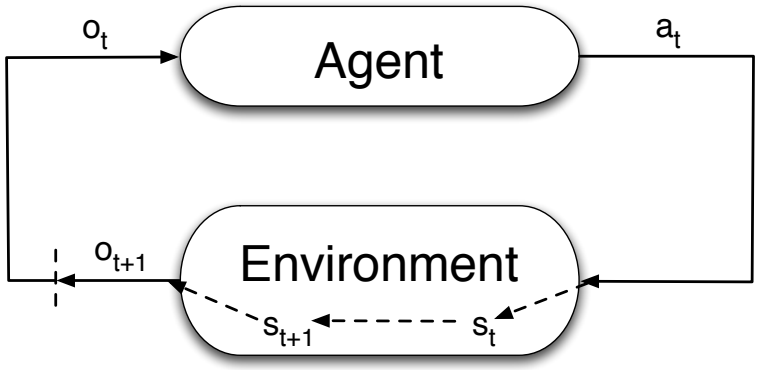


Figure 2.1: The agent-environment interface. At time step t the agent selects action a_t. The environment changes state probabilistically from s_t to s_{t+1} depending on the action. An observation o_{t+1} is emitted back to the agent.

observation may be a vector of features (as seen in Chapters 4, 5, and 6). The world is partially observable when the observation is insufficient to identify the environmental state.

A Markov Decision Processes (MDP) is defined by a set of observable states, a set of actions, state transition probabilities, and reward probabilities (associated with each state transition). A partially observable MDP (POMDP) is defined similarly to an MDP, but the states are not observable; instead, there is a distribution of observations (or feature vectors) associated with every state. The definition of MDPs and POMDPs are similar to the DDS paradigm described above, with the exception being that reward is not modeled in a DDS. Both MDPs and POMDPs generalize to a DDS in which reward is simply treated as an element in the feature vector, receiving no special distinction from any other observation.

2.2 Temporal-difference methods

TD methods are a class of algorithms that measure predictive error as the difference between temporally successive predictions (Sutton, 1988). The TD approach to learning contrasts with Monte Carlo approaches which measure predictive error as the difference between the current prediction and the final outcome of a behavior. These two classes of learning algorithms can be viewed as existing on a single continuum. On one end is single-step TD learning (TD(0)), where the prediction at time $t + 1$ is used as a predictive target for the prediction at time t. Monte Carlo algorithms occupy the opposite end of the spectrum, using the final outcome at time T as the target for the prediction at time t. With TD learning, predictions can be updated immediately whereas with Monte Carlo learning, predictions

cannot be updated until the final outcome is observed. Between Monte Carlo and single-step TD learning are algorithms that blend predictive targets of different lengths. A notable algorithm that bridges TD(0) and Monte Carlo is TD(λ), where λ an exponential weighting scheme that combines the predictions at time $t+1, t+2, \ldots, t+n, n \leq T$ such that lower values of λ place heavier weight on events closer in the future and higher values of λ place heavier weight on more distant outcomes.

TD learning has been used to solve reinforcement learning problems (problems in which an agent seeks to maximize expected reward, e.g., MDPs). TD agents find optimal policies in MDPs by learning expected rewards, and selecting the action with the highest expected reward in each state (Sutton & Barto, 1998). Problems from elevator scheduling (Crites & Barto, 1996) to learning to play backgammon (Tesauro, 1995) have been framed as reinforcement-learning problems which can be solved with TD learning.

Temporal-difference methods can predict quantities other than reward. TD methods have been used to predict state, effectively using TD algorithms to construct a model of the world (Sutton, 1995). Sutton, Precup, and Singh used temporal difference methods to model state and reward for *options*—temporally extended actions (1999). This thesis presents several algorithms based on the options framework. A formal description of the framework is provided in Chapter 4.

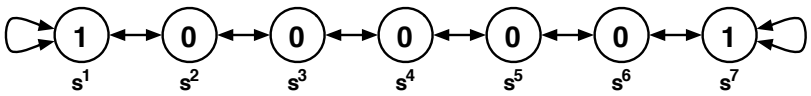
Temporal-difference Networks

As mentioned in the first chapter, a TD network is actually two conceptually separate networks: the question network and the answer network. The question network specifies the targets of learning; the answer network learns and computes predictions. TD networks permit the compositional specification of learning targets so that predictions can be made about other predictions.

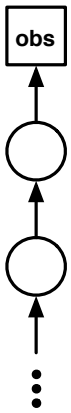
Each node in a TD network attempts to learn the expected value of its target as specified by the question network. This target may either be the value of another node on the next time step or an observation on the next time step. The target relationship is atypical for TD learning because the target is a different prediction; in typical TD learning a prediction targets itself on a future time step.

A gradient-descent learning rule is applied in the answer network to learn a set of weights that allow the agent to generate predictions from the previous time step's predictions.

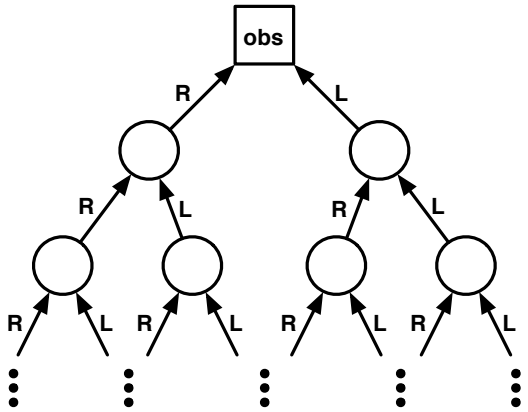
Sutton and Tanner conducted a suite of experiments, using the 7-state environment



(a) In the 7-state environment the agent observed either both the value (0 or 1) and the state label (s^i) or only the value.



(b) Action unconditional



(c) Action conditional

Figure 2.2: The temporal-difference network algorithm learned the correct predictions about the 7-state environment pictured in (a). Predictions were learned unconditional of actions (b) and action-conditional (c). (These figures originally appeared in Sutton & Tanner (2004))

pictured in Figure 2.2a as a testbed for their new algorithm (2004). The agent transitions between states by choosing either the left (L) or right (R) action. In each state the agent observes 0 or 1 depending on its current state. In the first set of experiments the agent had access to a label (s^i), which uniquely identified each environmental state. The agent learned predictions for two different TD networks by training on a sequence generated by a random walk. The first network, pictured in Figure 2.2b, made predictions about the observation n steps in the future (by using a chain of n nodes). The second network, pictured in Figure 2.2c, made all action-conditional predictions of length n and less (predictions about the observation bit for all action sequences of length n and less). Both networks were shown to make better predictions than a Monte Carlo algorithm.

A third experiment explored the partially observable case. Instead of state labels, the agent observed 1's in the ouside states and 0's in the interior states (see Figure 2.2a). The predictions of an action conditional TD network (Figure 2.2c, but with four levels) were learned from experience and were used to represent state. In these experiments the one-step error[1] approached 0 over time.

[1]The error was computed by comparing the one-step prediction to the actual value observed on the next time step.

The TD-network framework was extended to increase representational power and data efficiency. Two existing extensions to the framework are discussed briefly here (Tanner presented a detailed description of the original TD network architecture and these extensions in his thesis (2005)). A third extension, including temporal abstraction in TD networks, is the subject of this thesis.

Certain worlds, despite being representable, could not be learned with the original TD network learning algorithm. This learning problem was overcome by modifying the TD network's state representation (Tanner & Sutton, 2005b). In the original framework, the TD network formed its state representation from predictions and observations. In the extended framework, the representation was augmented with history information in order to assist learning. The effectiveness of the new algorithm was demonstrated on a 104-state grid world.

A second extension, TD(λ) networks, augmented the TD network learning algorithm by implementing inter-node eligibility traces (Tanner & Sutton, 2005a). TD(λ) networks were shown to learn correct predictions for worlds with a fraction of the data required by the original algorithm. Implementing the traces incurred minimal computational overhead.

In this thesis an additional extension generalizes the TD-network framework to incorporate temporal abstraction. Rather than making predictions about one-step actions, the augmented TD-network framework predicts the outcome of extended behaviors.

2.3 Grounded Models

Temporal-difference networks model the world with action-conditional predictions about observations. The approaches to modeling DDS's in this section share a common aesthetic and are thus said to be *grounded* models. Grounded (or *experience-oriented*) models are desirable because they are often easier to learn from data than a latent (or hidden) state model. Grounded models do not attempt to hypothesize the existence of an underlying environmental state, rather state is constructed from and represented as an agent's observables.

2.3.1 Diversity-based Inference

One of the inspirations for predictive representations was *Diversity-based Inference of Finite Automata*, in which the structure of a deterministic finite-state automata was inferred from data (Rivest & Schapire, 1993). A finite-state automaton can be described as a DDS with deterministic transitions. Rivest and Schapire introduced the notion of a *test*: a set of

actions followed by an observation $(a_1, a_2, \ldots, a_n, o_n)$. A test succeeds if the agent, starting from a given state, follows the sequence of actions specified by the test and observes o_n at the end of the trajectory. The goal of an agent is to construct a *perfect model* of its environment—that is, to know every test's probability of success.

Tests were divided into equivalence classes in which two tests were equivalent if, from every state in the environment, the two tests made the same predictions as each other. The equivalence classes over tests were used to construct an *update graph*: a graph in which equivalence classes corresponded to a vertices, and actions corresponded to edges. An agent has a perfect model of a world if it has an update graph and test values from each equivalence class because the update graph specifies how equivalence classes are connected by actions and the test values imply that the agent knows the outcome of transitioning between equivalence classes.

Rivest and Schapire presented algorithms that build the update graph and place the learner in a state for which the result of all tests is known, thus learning a perfect model of the environment. Initially, these algorithms required an oracle to determine whether tests were equivalent, but later algorithms determined test equivalence from data.

Hundt, Panagaden, Pineau, and Precup (2006) developed a theoretical framework for modeling DDSs (which could be considered an extension of Rivest and Schapire's work to stochastic systems). Hundt et al. presented the idea of creating a dual and double-dual representation of a DDS (or equivalently a POMDP). Their dual representation is to a DSS as Rivest and Schapire's update graph is to a finite-state automaton. Tests are generalized into *experiments*, a non-empty sequence of tests. As in Rivest and Schapire's work, equivalence classes are defined, now over experiments rather than tests, and a structure similar to an update graph can be constructed. A new set of experiments are defined in the dual which allows the construction of a double-dual representation, a representation that is equivalent to the original DDS in its most compact form.

2.3.2 Predictive State Representations

Predictive state representations (PSRs), are a class of predictive models that are based on the principle that the state of an unknown system can be modeled as a set of predictions about future interactions with the world. Since their introduction in 2002, PSRs have been the subject of a considerable amount of research. This section traces chronologically through the evolution of PSRs, ending with work that combines PSRs and options—most closely

resembling the integration of temporal-difference networks and options presented in this thesis.

In 2002, Littman, Sutton, and Singh introduced PSRs—an approach to modeling dynamical systems influenced by Rivest and Schapire's work (1993). Littman et al. redefined a test as a sequence of action-observation pairs of the form $a_0, o_0, a_1, o_1, \ldots, a_n, o_n$. The value of a test is the probability that the agent will observe $o_0, o_1, \ldots, o_n$ if it takes the actions $a_0, a_1, \ldots, a_n$ (ie. $Pr(o_i = o_0, o_{i+1} = o_1, \ldots, o_{i+n} = o_n; a_i = a_0, a_{i+1} = a_1, \ldots, a_{i+n} = a_n)$). Littman et al.'s premise was that knowing the value of all possible tests is equivalent to complete knowledge of the world. They went on to show that the value of all possible tests could be computed from a set of linearly independent tests—the probability distribution over all possible futures can be computed from a finite set of tests. Furthermore, they provide a proof by construction that any finite POMDP can be converted into a linear PSR where the number of linearly independent tests will be less than or equal to the number of underlying states in the POMDP model.

Singh, Littman, Jong, Pardoe, and Stone introduced the first learning algorithm for PSRs (2003). The first use of the term *core tests*, a set of tests from which all other tests can be computed (i.e., the linearly independent tests described in the previous paragraph), is found in this work. Singh et al. observed that in order to update the core tests it was also necessary to maintain predictions for all the 1-step tests, called *extension tests*. The values of core tests are updated by a projection vector learned via a gradient-descent learning rule.

James and Singh presented a second learning algorithm for PSRs (2004). This algorithm modeled systems with a reset action and, in addition to updating predictions, discovered a set of core tests. A major contribution of this work was the introduction of the *history-test prediction matrix* (which would later be called the *system-dynamics matrix*), an infinite matrix whose rows correspond to all possible histories and whose columns correspond to all possible tests. Elements in the matrix represented the prediction for a test given a history. The algorithm worked by first considering a sub-matrix of the system-dynamics matrix with only the one step tests (one action-observation pair). The sub-matrix was populated through the agent's interaction with the environment until each test had been sampled a minimum number of times. The reset action was a key part of this sampling process because it allowed the agent to return to the null history and thus receive multiple samples for each history-test combination. The rank of the sub-matrix was then estimated. (The rank is equivalent to the number of linearly independent tests (i.e. core tests) in the sub-matrix.) A new sub-matrix

with all the two step tests was then considered and the sampling process was repeated. The sub-matrix was expanded on each iteration of the algorithm, increasing the length of tests by one until the rank of the sub-matrix did not increase between iterations. At this point the algorithm had found a set of core tests from which all predictions could be computed. A set of parameters used to keep core tests updated could then be computed from data.

Around the same time as James and Singh's learning algorithm (2004), Rosencrantz, Gordon, and Thrun introduced the *transformed* predictive state representation (TPSR) algortihm (2004). TPSRs differed from PSRs because they did not seek a minimal set of core tests. Instead, an agent learned about a large number of tests, which were then projected to a low-dimensional space. The agent used the transformed predictions in the low-dimensional space as features in its representation.

On the theory front, Rudary and Singh introduced a formalism for non-linear PSRs (EPSRs) (2004). The new formalism was based on *e*-tests, which, like the tests of Rivest and Schapire (1993), were defined as a sequence of actions followed by an observation. EPSRs could be exponentially smaller than equivalent POMDP or linear PSR models. In another paper on PSR theory, Singh, James, and Rudary further formalized the system dynamics matrix and demonstrated the generality of PSR models (2004). They showed that while a PSR can model any system representable by a POMDP, there exist systems that can be modeled by PSRs that cannot be modeled by a POMDP.

Related to PSRs are Observable Operator Models (OOMs) which model a time-series of observations, generated by an unknown stochastic process as a sequence of operators (Jaeger, 1998; Jaeger, 2000). In Jaeger's model, the sequence of observations can be interpreted as a series of actions taken by the unknown process and thus the observations are both observable quantities and operators. Formally, an OOM is described by a set of matrices (each corresponding to an observable operator) and a starting vector. The relationship between OOMs and PSRs is discussed by Singh, James, and Rudary (2004).

Also in 2004, James, Singh and Littman presented an application of PSRs to the control problem. Two new algorithms were proposed in this work: the PSR incremental pruning (PSR-IP) algorithm and a Q-learning algorithm for PSRs. The PSR-IP algorithm is a direct adaptation of a POMDP learning algorithm in which a piecewise-linear value function is incrementally improved by preserving the best pieces on each iteration (Cassandra, Littman, & Zhang, 1997). Q-learning with PSRs was carried out by discretizing the continuous-valued prediction vectors. Multiple tilings, each offset by a small amount, were defined over

the prediction space, and the index of the tile occupied by the prediction vector in each overlapping tiling was used as a feature by the Q-learning agent.

Two separate algorithms were introduced in 2005 that allowed PSRs to be learned in the absence of an action that resets the agent back to a known state. Wolfe, James, and Singh introduced the suffix-history algorithm that removed the need for a reset action by instead considering history suffixes as rows in the system-dynamics matrix (2005). For example, a length 3 history $h = a_0, o_0, a_1, o_1, a_2, o_2$ could be used to update the rows corresponding to h, $h' = a_1, o_1, a_2, o_2$ and $h'' = a_2, o_2$. The algorithm considers sub-matrices of system-dynamics network as in James and Singh (2004), but now the sub-matrix being considered on iteration n will have all the n-step tests as columns and all the n-length histories as rows. In addition to removing the need for the reset action, Wolfe et al. implemented the first temporal-difference approach to learning PSR models.

A reset-free, on-line algorithm for learning PSR models was also introduced by McCracken and Bowling (2005). McCracken and Bowling limited the number of histories that the agent could remember so that the oldest history was forgotten when a new data point was encountered. A new row corresponding to the latest data point was then added into the approximated system-dynamics matrix. Regression was performed on the approximated matrix to extract the parameters of the PSR model. McCracken and Bowling's also proposed a new approach to discovering core tests. A new matrix was formed by appending the column corresponding to a non-core test to the approximated the system-dynamics matrix. If the condition number[2] of the new matrix surpasses a particular threshold then the new test was likely to be linearly independent from the current set of core tests and thus should be included to the set of core tests.

The memory-PSR (mPSR) model was introduced by James, Wolfe, and Singh in 2005. They partitioned the system-dynamics matrix according to histories, each partition forming a sub-matrix with its own set of core tests and parameters. James et al. proved that the size of the mPSR model was at most the number of partitions times the size of the PSR model, since in the worst case each partition had as many core tests as the full system; however, it was often the case that the mPSR model was more compact than the equivalent PSR model. Another contribution of this work is the identification of *landmarks*—memories which completely identify the current state. James and Singh used the mPSR model in the context of planning—they implemented the mPSR-IP algorithm which was shown to

[2]The ratio between the largest and smallest singular values of a matrix.

outperform both the PSR-IP (mentioned above) and POMDP-IP algorithms in most test problems (2005).

PSRs and the domains they could model were further formalized in the paper *Learning Predictive Representations from a History*, in which the complexity of both the environment and the agent was defined as the number of core tests needed to represent both the PSR and the agent's policy (Wiewiora, 2005). An interesting insight in this work is that actions and observations in tests can be reversed to form a-tests (a sequence of observations and actions $o_0, a_1, o_1, a_2, \ldots, o_n, a_{n+1}$) and the complexity of an agent's policy could be defined by finding the set of core a-tests. A *regular form* PSR is defined in this work as a PSR with a minimal set of core tests where each core test is either the empty test or an extension of a core test. Wiewiora further showed that any PSR can be converted into a regular form PSR with an equivalent or smaller number of tests.

Another new development in PSR literature was the work of Bowling, McCracken, James, Neufeld, and Wilkinson (2006). Until this work, PSRs were learned from blind policies—policies that were independent of the observations (i.e., $\pi(\cdot, a)$). All prior PSR learning algorithms were only guaranteed to learn a correct model if the learning agent followed a blind policy. Bowling et al. presented a new learning algorithm that allowed the agent to learn correct predictions even when following a non-blind policy. They also introduced a new exploration algorithm that took advantage of a non-blind policy to collect data more efficiently.

A recently developed offshoot of PSRs are Predictive Linear-Gaussian (PLG) models, first introduced by Rudary, Singh, and Wingate (2005). While PSRs model discrete dynamical systems, PLGs extend predictive representations to uncontrolled domains (no actions) with continuous observations. PLGs have been extended to use kernel methods (Wingate & Singh, 2006a), to model systems as a mixture of PLGs (Wingate & Singh, 2006b) and to incorporate actions into the model (Rudary & Singh, 2006).

PSRs and Options

In 2006, Wolfe and Singh presented *Predictive State Representations with Options*—the work in the literature most closely related to this thesis. Wolfe and Singh's framework combines options and PSRs by maintaining PSR models at two time-scales: the primitive action time-scale and the option time-scale. Option tests are defined as a sequence $\omega_0 o_0 \ldots \omega_n o_n$ where ω_i is an option followed until termination, and o_i is the observation at termination. The

definition of option tests is similar to that of traditional tests except that primitive actions are replaced by an option's policy.

Each option has a corresponding system-dynamics matrix (in which each column corresponds to a primitive test) whose entries are updated whenever the option is being executed. The entries of an option-level system-dynamics matrix (in which each column corresponds to an option test) are updated after an option terminates. Each system-dynamics matrix can be modeled by a PSR and thus the previously mentioned PSR learning algorithms could be implemented to learn the model parameters. The option-level PSR can be learned with any of the reset-free algorithms described above; the action-level PSRs can be learned with any algorithm (including those with reset because each option initiation occurs from the null-history). Wolfe and Singh referred to the algorithm that simultaneously learns the action-level and option-level PSRs as the *Hierarchical* PSR (HPSR) algorithm.

The HPSR algorithm was used to model two domains: a 78-state grid-world and a 500-state grid-world. The 78-state world consisted of 9 rooms connected to a central hallway. The simple actions available to the agent were the cardinal directions (North, South, East, and West); the options available to the agent permitted it to move directly between rooms (60 options provided in all) The 500-state world was a modified version of the Taxi domain (Dietterich, 1998) in which the agent could move in the four cardinal directions in a world with 25 grid cells. Four special locations were identified in which a passenger could either be picked up or dropped off. The agent transported passengers between the pick-up point and the drop-off point. Options were provided for picking up a passenger, dropping off a passenger and navigating between the special locations (14 in total). In both domains, the HPSR agent learned a model with low prediction error in less computational time than a linear PSR agent.

2.3.3 History-Based Representations

Like other models mentioned in this chapter, history-based representations are grounded in an agent's actions and observations. The simplest history-based models are Markov-k models. In a Markovian system, the observation uniquely identifies the agent's position in the world; in a Markov-k system, knowledge of the past k action-observation pairs identify state. A Markov-k model represents state with the past k action-observation pairs.

Variable-length memory models are more sophisticated than Markov-k models in that different lengths of history represent different states. A longer history can be used to rep-

resent situations that need a finer grained distinction. Ring (1994) and McCallum (1996) modeled environments with a combination of variable-length histories and reward signals as a state representation.

2.3.4 Schemas

Drescher's *schema* learning (1991) is another grounded approach to building a predictive model. The goal of the learning agent is to learn the effects of actions in the world. There is an underlying assumption that there is regularity in the world—taking certain actions in certain situations will lead to a specific result—that can be captured by a schema model. Formally, a schema is composed of three components: a context, an action, and a result. More plainly, a schema is an action-conditional prediction about the next observations (result), given that the current observations were in some configuration (context).

To deal with hidden state, schemas can propose *synthetic items* which are elements that the agent adds into the observation vector. The value of these new elements is learned by the agent. If a schema is not reliable for some context, action and result, then the agent supposes the existence of a synthetic item that can make the schema's prediction true.

Holmes and Isbell revisited Drescher's work and extended schemas to handle discrete observations (2004) (Drescher's schemas handled only binary observations). The learning algorithm is also modified to handle stochastic domains. Schemas are shown to achieve a similar error measure to PSRs on sample domains with much less training data.

2.4 Discussion and Conclusions

This section presented the class of environments that are modeled by TD networks (and the extended TD networks presented in Chapters 4, 5, and 6). TD networks belong to a larger class of models called predictive representations in which knowledge is represented as predictions about possible future experience. In turn, predictive representations belong to the larger class of experience-oriented models which relate knowledge to an agent's experience, both historical and future.

Closely related to TD networks are PSRs; closely related to this thesis are PSRs with options. This chapter traced through the existing PSR literature from their first appearance in literature to their latest developments, ending with a description of the HPSR algorithm—the algorithm that allows an agent to model both action-level and option-level PSRs.

Chapter 3

The Predictive Representations Hypothesis

The experiments in this chapter are designed to test the *predictive representations hypothesis*, which holds that particularly good generalization will result from representing the state of the world in terms of predictions about possible future experience.[1]The abstract state representation constructed from TD network predictions is a *generalization* of the environmental state—states with similar predictions are treated as a single abstract state. A grid-world navigation problem is used as a milieu for testing the hypothesis. Experiments in this chapter compare the performance of reinforcement learning agents with state representations constructed from:

- the predictions of a TD network,
- the environmental state,
- history.

A large portion of current predictive representation research explores representation acquisition (Singh, Littman, Jong, Pardoe, & Stone, 2003; Sutton & Tanner, 2004; James & Singh, 2004). However, employing predictive representations in control problems is beginning to be explored as well (James, Singh, & Littman, 2004; James & Singh, 2005). In this chapter, a TD network's predictions are used as a state representation for a reinforcement learning task. The prediction-based reinforcement learning agent is shown to learn a near-optimal solution to the navigation problem with less training than the other agents.

[1]Portions of this chapter originally appeared in the proceedings of the 2005 International Joint Conference on Artificial Intelligence (Rafols, Ring, Sutton, & Tanner, 2005), however the majority of this chapter is original work.

Confounding Factor	Solution
• Representation acquisition	• An oracle provides predictions
• Function Approximation	• Tabular predictive representations
• Environment Complexity	• Deterministic transitions and observations in the grid world

Figure 3.1: The three main confounding factors and the corresponding solutions.

3.1 Motivation and Confounding Factors

A good generalization captures underlying regularities of the environment, increases an agent's ability to receive reward, and accelerates learning. Good generalization often occurs when situations that require a similar response are grouped together because learning in one situation will transfer to all other situations in the group. TD networks are expected to usefully generalize the state space because situations in which action sequences lead to similar outcomes will have similar representations.

There are several confounding factors that make the predictive representations hypothesis resistant to testing. In order to test the hypothesis as directly as possible, steps were taken to control for these confounding factors:

- Evaluating the quality of a representation is difficult when an agent tries to simultaneously accomplish a task in the environment and learn a TD network's predictions. Rather than learning the predictions, an oracle provides the agent with correct predictions.

- The predictions of a TD network are generally used as the features of a function approximator, bringing up issues in function approximation. To control for this, a tabular state representation is constructed from the TD network's predictions (Section 3.3).

- Stochasticity in an environment's dynamics may lead to a large amount of variance in what an agent learns. This issue is controlled for by conducting experiments on an environment with deterministic transitions and observations.

Figure 3.1 summarizes possible confounding factors and the solutions that control for these factors.

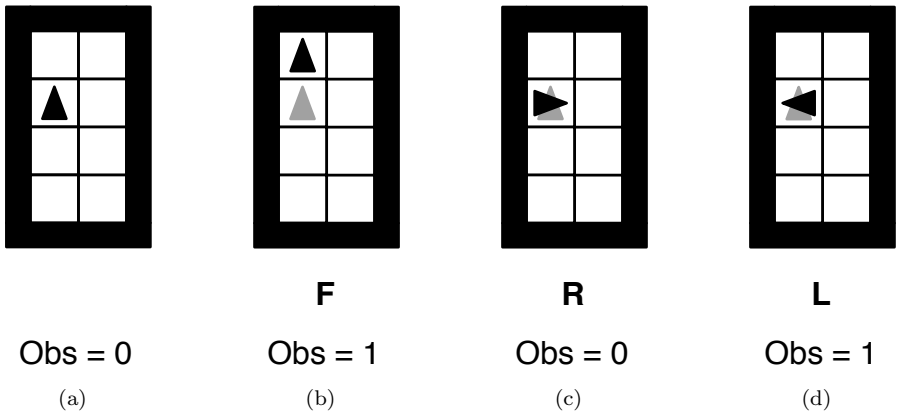


Figure 3.2: A small example grid-world. From its initial position (a), the agent can take one of three actions: step forward (b), rotate 90° right (c), or rotate 90° left (d). The agent observes a 1 if is immediately facing a black grid cell and a 0 otherwise. In Figure a) the agent's observation is 0. In b), c), and d), the observation is 1, 0, and 1, respectively.

3.2 Agent and Environment

Throughout this work experiments are conducted in a grid world with an *egocentric* agent—all actions taken and observations received are in relation to the direction that the agent is facing. The agent observes a single bit, indicating whether the agent is facing a wall (black grid-cell). The actions available to the agent are: step forward one grid cell (F), rotate 90° right (R), and rotate 90° left (L). Figure 3.2 illustrates the physics of the world. If the agent is facing open space (i.e., the observation is 0), the step forward action moves the agent one grid-cell in its direction; if the agent is facing a wall (i.e., the observation is 1), the step-forward action has no effect. The rotate actions spin the agent either 90° clockwise (R) or 90° counter-clockwise (L). As mentioned in Figure 3.1, all actions are deterministic.

Throughout this thesis, a unique labeling assigned to each combination of grid cell and direction will be referred to as the agent's *environmental state*. Typically, the agent does not observe the environmental state. Rather, the agent observes a feature vector in each environmental state.

3.3 Tabular Predictive Representations

As described in Figure 3.1, there is a need to control for representation acquisition and function approximation. The construction of *identically predictive classes* removes these two potentially confounding factors from consideration. In the following section, we explain how a TD network's predictions are converted into a tabular representation.

A *test* of length m is defined as a sequence of actions $a_1 a_2 \ldots a_m$ followed by a single observation. The action-conditional TD network pictured in Figure 2.2c exhaustively enumerates all tests of length $\leq n$ (where n is the number of levels in the network). This TD network structure is used to specify the agent's predictions. In total, there are $N = \sum_{i=1}^{n} |a|^i = |a|^{n+1} - 1$ tests where $|a|$ is the number of actions available to the agent (as specified in Section 3.2, $|a| = 3$ in our experiments).

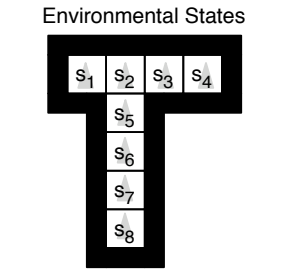
A *configuration* is the set of all predictions in the TD network at a specific time step. Because each test has a binary outcome (the agent either will or will not observe a wall at the end of a test), there are 2^N possible configurations. If two environmental states cannot be distinguished by any of the N tests, then the configuration is identical in both states, and these states are said to be *identically predictive* for the n-level TD network. Environmental states can thus be grouped into c classes in which each class contains all states with identical configurations. The classes are labeled 1 through c and the agent observes the class label of the environmental state that it occupies.

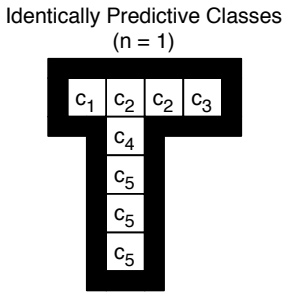
Other researchers have presented work in which predictions were used to define classes. Rivest & Schapire similarly defined a set of equivalence classes for $n = \infty$ (1994); Hundt, Panagaden, Pineau, & Precup generalized the equivalence classes to be over sequences of tests (2006).

A graphical representation of the process of identifying identically predictive classes is shown in Figure 3.3. This example shows the predictive classes for $n = 1$ when the agent is facing North. Each grid cell in the environmental state has a unique label (s_1 to s_8). The three middle columns of the table contain all one-step predictions at each environmental state. Certain states are identically predictive since all three predictions are the same. The identically predictive states are all given the same class label, c_1 to c_5.

In general, as the length of tests (n) increases, both the number of tests (N) and the number of identically predictive classes (c) increases. There are fewer states per class on average and thus the agent's representation of its environment becomes more expressive. Despite the fact that the number of configurations grows exponentially with n, the number of classes tends to increase quite slowly in environments with even a moderate amount of regularity.

In the limit, c will no longer increase for any value of n, meaning that no additional prediction can distinguish between the environmental states belonging to a predictive class. It is at this point that the identically predictive classes represent a *sufficient statistic*.





Environmental State	Predictions (n = 1)			Predictive Class
	L	F	R	
s_1	1	1	0	c_1
s_2	0	1	0	c_2
s_3	0	1	0	c_2
s_4	0	1	1	c_3
s_5	1	1	1	c_4
s_6	1	0	1	c_5
s_7	1	0	1	c_5
s_8	1	0	1	c_5

Figure 3.3: Grouping environmental states into identically predictive classes. The leftmost column of the table contains the unique labelings of the environmental states $(s_1, \ldots, s_8)$. The middle three columns show the predictions for the rotate-left, step-forward, and rotate-right actions. The rightmost column shows the predictive class that each environmental state falls into $(c_1, \ldots, c_5)$. Notice that all environmental states in a predictive class have an identical set of predictions.

3.3.1 Sufficient Statistics

If the sufficient statistic has C classes then it is impossible for any test to distinguish between the environmental states belonging to any predictive class (and therefore, new predictive classes cannot be formed). If additional predictions could be used to distinguish a new class, this would imply that the representation with C classes is not a sufficient statistic since further distinction is possible.

The environmental state represents a sufficient statistic for the environment, but this representation is not necessarily a *minimal* sufficient statistic. Consider Figure 3.4 for example. The grid world consists of 17 grid cells, with four possible agent orientations in each cell. There are this 68 environmental states, and knowledge of the environmental state summarizes all past history (and therefore the environmental state is a sufficient statistic). An egocentric agent, as described in Section 3.2, will be unable to distinguish between the four arms of the cross as all predictions (even those of infinite length) will be identical for each arm given the large amount of symmetry in the environment. The agent will be able to identify at most 17 distinct predictive classes: a class for each of the four different orientations in each of an arm's four grid cells and one class for the center square which is

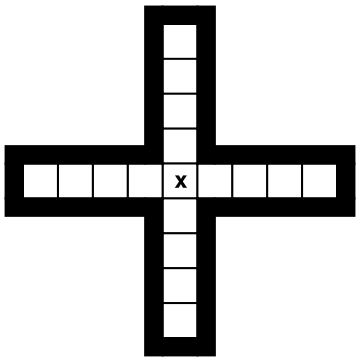


Figure 3.4: An example cross-shaped grid-world in which the predictive representation is aliased. There are four orientations for each grid cell meaning that there are 68 distinct environmental states. A predictive representation will identify (at most) 17 identically predictive classes.

identically predictive regardless of orientation.

An advantage of representing state with predictions is evident if the agent is tasked with learning a path to the center square of Figure 3.4 (marked by the **x**). An agent with a predictive representation would learn to solve the task more quickly that an agent that observes the environmental state because there are fewer unique classes than environmental states (17 predictive classes vs. 68 environmental states).

3.3.2 Performance and Generalization

The performance of a reinforcement-learning agent in an episodic task can be quantified by the total amount of reward received per episode (the reward received between starting the task and reaching the goal). In the limit, as the amount of training time goes to infinity, an agent with access to the environmental state can learn an optimal policy. However, as environments grow arbitrarily large, learning an optimal (or even near-optimal) policy becomes impractical because the agent must learn the value of every action in every environmental state. A representation that generalizes well can reduce the size of the state space and accelerate learning.

As a generalization is broadened and the amount of state abstraction is increased, asymptotic performance is traded for speed of learning. As discussed in the previous section, as the length of tests n increases, more predictive classes c are distinguished and there are thus fewer environmental states in each predictive class. With shorter tests, there are fewer classes to learn about, but there is a risk of a situation where multiple environmental states within the class disagree on the optimal action. Consider the situation where the **x** is not

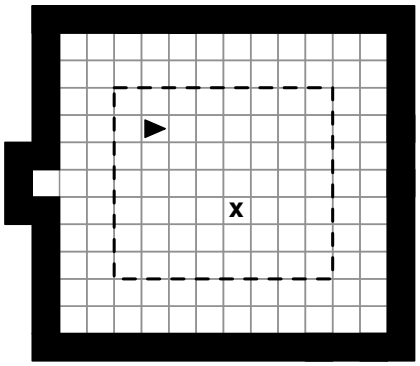


Figure 3.5: Disagreements on optimal action selection may occur in this grid-world because the grid cells within the dotted box will be grouped into a single identically predictive class if we only consider the 1-step tests. Depending on the agent's environmental state, the optimal action may be any of three possible actions. A more expressive representation would minimize action selection disagreements.

placed in the center cell of Figure 3.4, but rather in the middle of one of the arms. Because the agent cannot distinguish between any of the arms, the agent will have trouble learning a path to the **x**.

More concretely, consider two states s_1 and s_2 grouped into a single predictive class. It may be optimal to rotate right in s_1, but the optimal action may be to step forward in s_2; however, because the two environmental states belong to the same predictive class, the agent will be forced to make a suboptimal action selection in either s_1 or s_2. In the worst case, the disagreement may be so severe that the agent is unable to find a reasonable policy. An example of this situation is shown in Figure 3.5. If the agent represents the world predictively with 1-step tests, all interior squares appear identical. However, depending on the agent's environmental state, the optimal action may be any of the three possible actions: step forward, rotate right or rotate left. All environmental states inside the dotted square appear the same to the agent and thus a single action must be mapped to the abstract state. Given a more expressive representation, the agent would be able to distinguish its position (by the number of step-forward actions taken before a wall is observed) and direction (by the number of rotations needed to be facing the notched wall).

An ideal tabular predictive representation will balance between learning rate and asymptotic performance. Thus, a value of n that has a small number of classes c, but also minimizes the number of policy-related disagreements is desirable.

3.3.3 Sarsa(0) with Identically Predictive Classes

All agents in this chapter were trained using the reinforcement-learning algorithm known as episodic tabular Sarsa(0) (Sutton & Barto, 1998). In the traditional Markov case—where the agent directly observes the environmental state—an action-value function is learned over the space of environmental states and actions. In this algorithm, the estimated value $Q(s, a)$ of each experienced state–action pair s, a is updated based on the immediate reward r and the estimated value of the next state-action pair; i.e,

$$\Delta Q(s, a) = \alpha[r + Q(s', a') - Q(s, a)],$$

where α is a learning-rate parameter.

Episodic tabular Sarsa(0) is implemented over the predictions by mapping environmental states to their corresponding identically predictive classes, as described in Section 3.3. The function $C(\cdot)$ provides this mapping, and the resulting classes are then treated by the Sarsa agent as though they were environmental states:

$$\Delta Q(C(s), a) = \alpha[r + Q(C(s'), a') - Q(C(s), a)] \tag{3.1}$$

Because no distinction is made between the states within a class, the learning that occurs in one environmental state applies to all states mapped to the same class.

3.4 Tabular History-based Representations

An approach to state representation related to predictive representations are history-based representations. Both representations relate the agent's location to sensations. Predictive representations identify state according to where the agent could go; history-based representations identify state according to where the agent has been. Fixed-length history approaches can easily be expressed in tabular form by labeling each k-length history. Tabular history-based representations are implemented as a point of comparison for tabular predictive representations.

As in Section 3.3.3, episodic tabular Sarsa(0) is implemented over the history-based representation. The function $H(\cdot)$ provides a mapping from each different k-length history to its label and these labels are treated by the Sarsa agent as though they were environmental states:

$$\Delta Q(H(o_0, \ldots, a_k, o_k), a) = \\ \alpha[r + Q(H(o_1, \ldots, a_{k+1}, o_{k+1}), a') - Q(H(o_0, \ldots, a_k, o_k), a)] \tag{3.2}$$

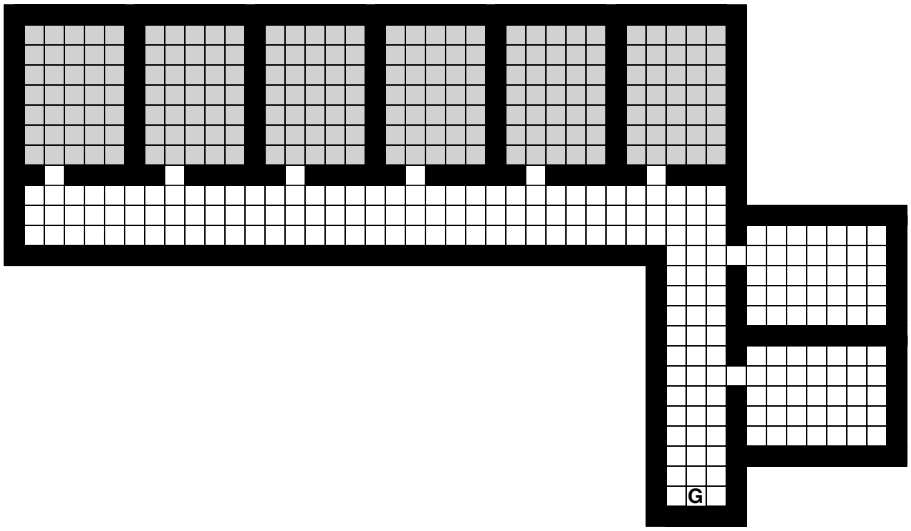


Figure 3.6: The "office" grid-world used for the navigation task. The agent starts an episode in one of the six top rooms (grey squares), and finishes in the square marked **G**.

3.5 Experiment Design

The predictive representations hypothesis was tested in the grid world shown in Figure 3.6. The environment was designed to resemble a typical office layout, and the task can be likened to finding the shortest path to the staircase. Many regularities exist in this environment (similar structure of rooms, uniform hallway width), thus representations that generalize well should allow their respective agents to exploit these regularities.

The dynamics of the agent and the environment are as described in Section 3.2. The rewards for the task are $+1$ for reaching the goal state (marked by **G**) and -1 on all other time steps. The environment has a total of 1696 states (424 grid cells and four orientations in each cell). The task is episodic; the agent is transported to a randomly chosen starting position in one of the top six rooms (the shaded cells) upon reaching the goal. Upon restart, the agent's action values are reset, and learning begins from scratch. On average, there are 42.2 steps along the optimal path from start to goal.

Actions were chosen according to an ϵ-greedy policy: with probability $1 - \epsilon$ the agent chooses the action with the highest expected reward, and with probability ϵ the agent chooses the action randomly. ϵ was set to 0.1 and α was set to 0.25—typical values for Sarsa agents carrying out episodic tasks in deterministic environments.

The experiments compared the performance of reinforcement-learning agents with three different state representations:

Agent		Unique Labels	% Environmental States
Markov	-	1696	100%
Predictive	n		
	2	67	3.9%
	3	185	10.7
	4	308	17.8
	5	416	24.1
	6	497	28.8
	7	568	32.9
Fixed-History	k		
	2	50	2.9%
	3	205	11.9
	4	790	45.7
	5	2,938	170.0
	6	10,660	616.9

Figure 3.7: The number of unique labels found in each of three state representations. n is the number of levels in the TD Network used to specify the predictions. k is the number of action-observation pairs in each history. The number of unique labels for the fixed-history representation is the number of different unique histories that appeared over the course of training. The amount of aggregation is the percentage of unique labels as compared to the number of environmental states.

- Markov state representation,
- tabular predictive representation (formed from the predictions of an n-level TD Network),
- tabular history-based representation (of length k).

The environmental-state agent observed the unique labeling of each environmental state; the predictive agent observed the predictive class label; the history-based agent observed the label associated with its k-step history. The number of unique labels for each different representation is shown in Figure 3.7. The amount of state aggregation that occurs in each method is shown in terms of the ratio of unique observations to environmental states.

An example of the classes identified from the predictions of a 1-level TD network is displayed in Figure 3.8 for when the agent is facing North. (Experiments were not conducted for $n = 1$, but the figure illustrates that each rooms share a common predictive structure)

3.6 Results

Performance results for agents with the three representations in Figure 3.7 are graphed in Figure 3.9. The data points used to generate the learning curves were the average number of steps per episode over the previous 10 episodes. The curves were averaged over 10,000 trials,

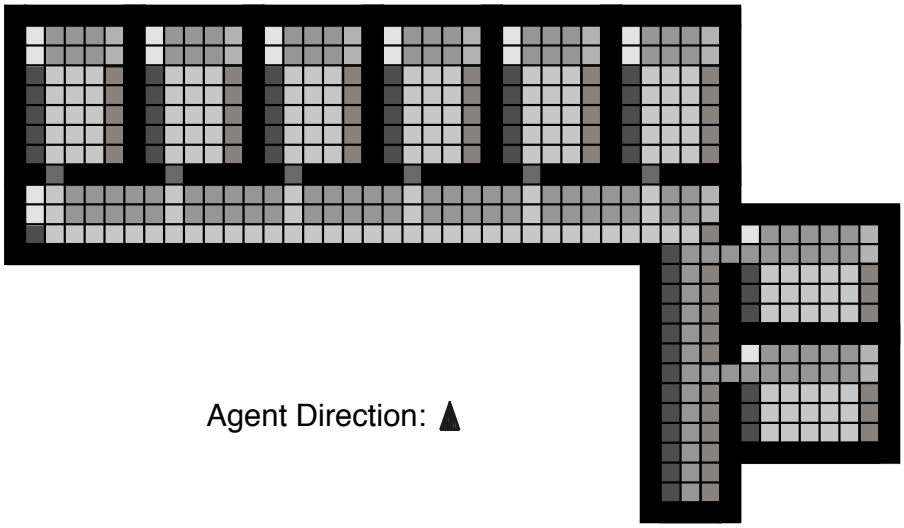
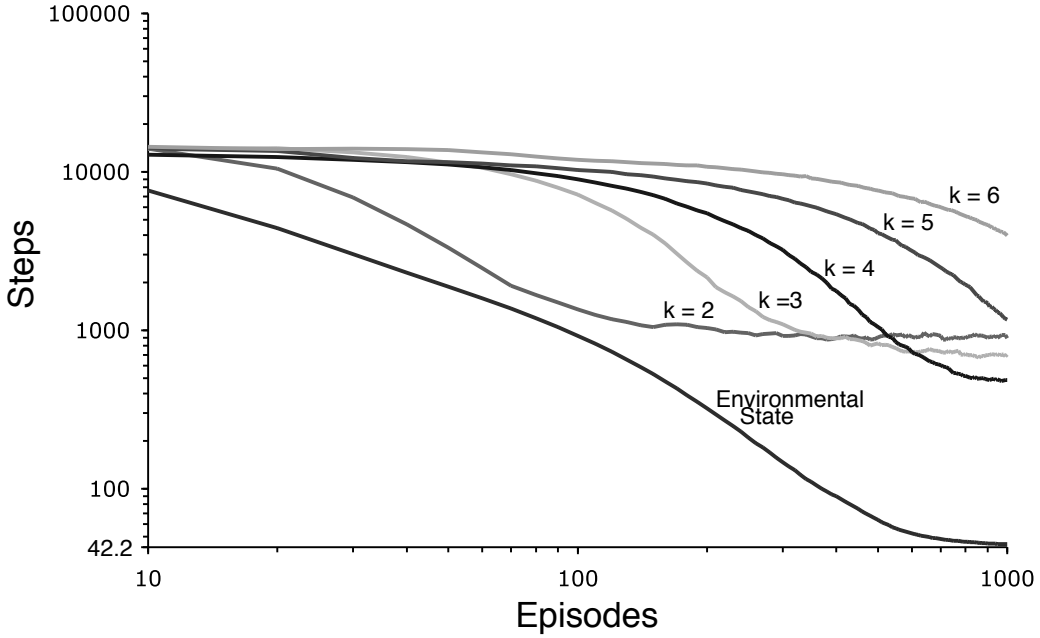


Figure 3.8: An illustration of the "office layout" grid-world divided into predictive classes for $n = 1$ when the agent is facing North. When $n = 1$, there are 7 identically predictive classes identified (out of 8 possible configurations). The classes are denoted by the different levels of shading.

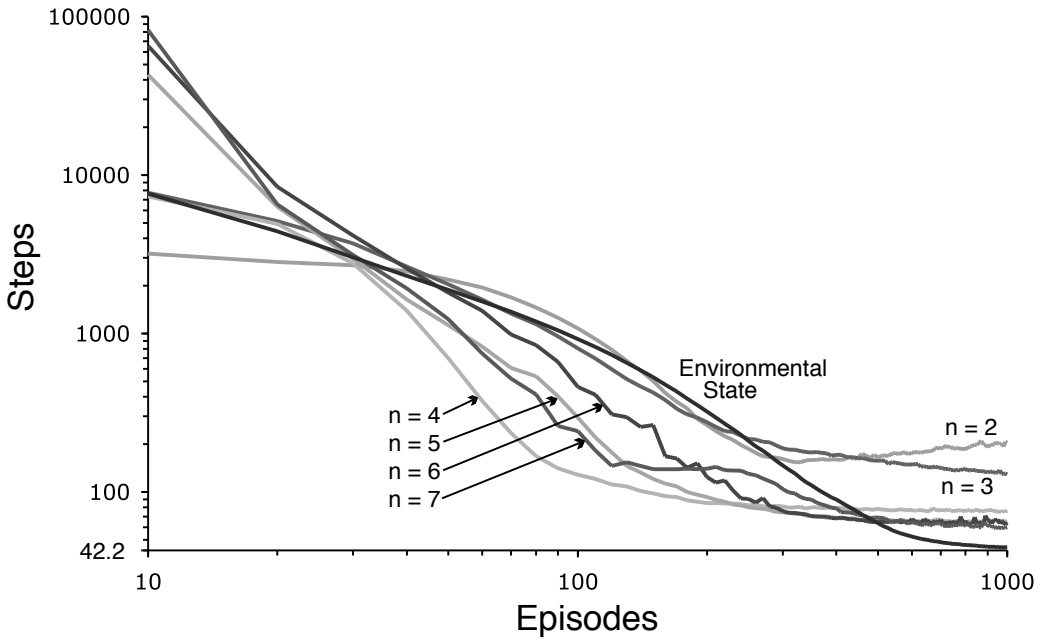
each trial lasting 1,000 episodes. Over the course of 1,000 episodes, the environmental-state agent showed a smooth, steadily improving curve, which by the $1,000^{th}$ episode is performing very close to optimal.

Figure 3.9a shows the learning rates for history-based representations with $k = 2$, 3, 4, 5, and 6 as compared to the learning rate of the agent that was provided with the environmental state. As k increased, the learning rate of the history-based agents decreased, but the asymptotic performance improved—a clear demonstration of the trade-off between representation expressiveness and learning speed. The number of histories increased (exponentially) with k, which negatively impacted learning speed but positively impacted the final results of learning.

Figure 3.9b shows the learning rates for predictive representations with $n = 2$, 3, 4, 5, 6, and 7 as compared to the learning rate of the agent that was provided with the environmental state. The results looked promising for predictive representations. They allowed both speedy learning and convergence to a good policy. In general, the results for the identically predictive representations were similar to those for the fixed-history representations in that convergence speed decreased and convergence quality increased as n increased. However, in contrast to the fixed-history representation, the number of identically predictive classes increased quite slowly with n (cf. Figure 3.7) and the generalization benefit of the predictive classes was clear. The representation effectively aggregated similar states, allowing the agent



(a) History-based representation performance graph.



(b) Predictive representation performance graph.

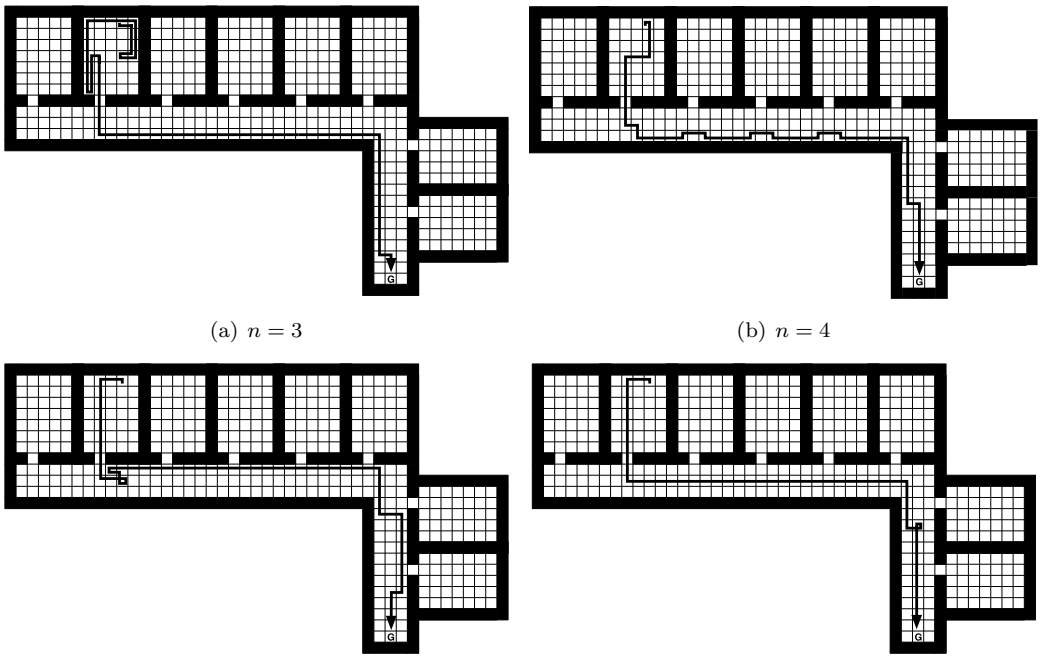
Figure 3.9: Performance graphs for a) history-based representations and b) predictive representations for various values of k and n as compared to performance with the environmental state. Notice that the scale on both axes are logarithmic.

to converge to near-optimal solutions.

In the $n = 2$ and $n = 3$ cases there was no improvement in learning speed, indicating there may be a maximum degree of state aggregation beyond which learning speed is not improved—an expected result as discussed in Section 3.3.2. The result for $n = 7$ is particularly interesting because early in learning (the first 100 episodes) the agent learned more quickly than in the $n = 5$ and $n = 6$ cases. However, learning plateaud for several hundred episodes before eventually surpassing all other values of n for the best asymptotic performance.

It can be argued that predictive representations fared better than history-based approaches due to the preprocessing used to create the identically predictive classes or that the naive approach to tabular histories could be improved upon. Both of these statements are true, but such arguments are tangential to the main purpose of the experiments: testing the predictive representations hypothesis. The crux of the experiments was testing whether predictive representations provide good generalization, and the experiments give credence to belief that they do indeed. For multiple values of n, predictive representations are shown to aggregate environmental states into predictive classes in such a way that learning is dramatically accelerated, while still finding reasonable solutions to the navigation task. The experiments with history-based representations show that beneficial generalization is not a property of experience-oriented representations in general, but a property of predictive representations.

Another possible objection to using predictive representations is that one could use a hand-coded mapping of states to classes or use some sort of heuristic for aggregating states into classes. These are both possible approaches to abstracting over state, but both approaches imply knowledge about the underlying state space and require external knowledge to be injected into the state representation. A key feature of predictive representations is that all knowledge can be acquired and verified by the agent itself. While, in our experiments, the predictions were provided by an oracle, ultimately it is hoped that the agent can learn the predictions from experience. The tabular predictive representation introduced in this chapter is based on the existence of environmental states that can be mapped to predictive classes. If the predictions were learned from data, then such a mapping would not need to exist. Instead, the generalization would appear naturally as a property of predictive representations—configurations of predictions would represent state and the existence of an underlying environmental state need not be considered.



(a) $n = 3$ (b) $n = 4$

(c) $n = 5$ (d) $n = 6$

Figure 3.10: Sample trajectories for various values of n.

Sample Routes

Four paths found by prediction-based agents (for $n = 3$, 4, 5, and 6) are shown in Figure 3.10. The agent was tested after a training run of 1,000 episodes. These representative routes were generated by the greedy policy—beginning from a fixed starting state, the action with the highest expected reward was selected in each predictive class along the path. (Note that the "greedy" path may vary dramatically between training runs for a fixed n.)

For $n = 3$, the agent appeared to have difficulty exiting the initial room and wasted many steps trying to find the exit. Once the exit was found, the agent took a direct path to the goal. The start of this path likely demonstrates a case where disagreements about the optimal action occurred (cf. Section 3.3.2) However, the hallway states were coarsely generalized allowing a straightforward path through the hall and to the goal.

When $n = 4$, the agent exited the room much more easily, but followed a less direct route upon reaching the hallway. Compared to $n = 3$, there were likely enough different predictive classes in the room to allow the agent to exit in a small number of steps. In the hallway, there was a visible perturbation when the agent was in line with each "doorway" (the single grid cell separating each room from the hallway). These grid cells evidently belonged to

common predictive classes as the agent followed the same sub-path at each doorway.

For $n = 5$, the agent exited the room easily and followed a relatively straight path through the hallway. However, the agent twisted twice upon exiting the room. With additional episodes the agent would likely learn to continue straight through rather than spinning.

Finally, in the $n = 6$ case, the agent found a direct route to the goal, the only mistake was a series of three left turns instead of a single right turn when approaching the goal. This error would also likely disappear with additional training. The mistakes made in the $n = 5$ and $n = 6$ cases demonstrate that, as n increased, the number of predictive classes grew, and thus more experience was necessary to fully learn the optimal action in each situation.

3.7 Discussion and Conclusions

The work in this chapter makes an initial attempt to demonstrate that representing the world in terms of prediction about possible future experience results in particularly good generalization. While the claim is broad and there are many possible confounding factors, this initial work lends weight to the possibility of a *yes* answer. In the presented experiments, tabular predictive representations were shown to generalize the environmental state space in such a way that the agent was able to learn a reasonable policy for a navigation task much more quickly than if provided with the environmental state.

As mentioned in Section 3.3.2, in certain environments there are possible goal locations for which a predictive representation-based agent would be unable to learn a reasonable policy. What would happen if the goal in Figure 3.4 were placed in the middle of one of the arms? Because all arms of the environment appear identical in a predictive representation, the agent would not be able to define a policy that consistently navigates the agent directly to the goal. This problem could be overcome by treating reward like all other observations and including it as part of the predictive state. A value function (learned by a reinforcement-learning agent) is a prediction of long-term future reward if actions are selected optimally. Treating reward as an observation allows an agent to make other predictions about expected reward such as predicting expected reward conditional on a specific sequence of actions.

The "officeworld" layout presented in this chapter is much larger in scale than any environment for which a predictive representation has been learned. The motivation behind most of this thesis is the desire to learn a representation for worlds the size of the "officeworld" and larger. Predictive representations have been demonstrated to usefully abstract the state

space—learning is accelerated by learning a policy over identically predictive classes. State abstraction is important when scaling to larger environments, but abstracting over time is equally important. In the following chapters, the incorporation of temporal abstraction increases the representational power of the TD-network framework.

Chapter 4

Augmenting Temporal-difference Networks with Options

This chapter presents the first[1] on-policy learning algorithm for option-conditional TD networks (OTD networks). This algorithm serves as a basis for the algorithms introduced in Chapters 5 and 6; these algorithms are the primary contribution of my thesis. Temporal abstraction is incorporated into TD-networks by extending the existing framework (Sutton & Tanner, 2004) to make long-term predictions. The inclusion of temporal abstraction is based on the options framework (Sutton, Precup, & Singh, 1999), an approach to temporal abstraction developed for reinforcement learning. Rather than conditioning predictions on actions which span a single time-step, predictions are conditioned on an option's policy and its termination condition. The agent learns predictions by following option policies until termination.

The chapter begins with a formal definition of both the TD-network framework and the options framework. This is followed by a description of the OTD network algorithm and a derivation that demonstrates that the forward and backward views of the new algorithm are equivalent. The chapter finishes with the presentation of experiments that suggest the correctness of the new algorithm.

[1]This chapter is based on work that appeared in the Proceedings of Advances in Neural Information Processing Systems 18 (Sutton, Rafols, & Koop, 2005). However, the algorithm introduced in this chapter is unique because it is strictly on-policy and thus the work found in this chapter is original.

4.1 Temporal-difference Networks

The previous overviews of TD networks (Section 1.2 and Section 2.2) are high-level descriptions and do not delve into the internal workings of the framework. This section presents a formal description of the TD network learning algorithm.

Temporal-difference networks are composed of a set of *predictive nodes* that are interconnected by two conceptually separate networks: the question network and the answer network. On each time step, t, the network makes n predictions: $\mathbf{y}_t = (y_t^0 \cdots y_t^n)^\top \in \mathcal{R}^n$.

The question network is specified by targets, $\mathbf{z}$, and conditions, $\mathbf{c}$. A prediction, y_t^i is the action-conditional expected value of a target:

$$y_t^i = E_\pi[z_{t+1}^i | c_{t+1}^i], \tag{4.1}$$

where π is the policy being followed (the behavior policy), z_{t+1}^i is the quantity being predicted and c_{t+1}^i indicates upon which action(s) the prediction is conditional. The target indicates what a node predicts—either an observation ($o \in \mathcal{O}$) or the value of another prediction (y^i where $0 \leq i \leq n$). The target is thus a mapping $z^i : \mathcal{O} \times \mathcal{R}^n \to \mathcal{R}$, and is defined as:

$$z_t^i = z^i(o_{t+1}, \tilde{\mathbf{y}}_{t+1}).^2 \tag{4.2}$$

The condition $c^i \in [0, 1]$, indicates to what extent the action taken at time t matches the action(s) on which prediction y^i is conditioned (typically, c_t^i is a binary variable).

The answer network learns the predictions which are computed as a function, $\mathbf{u}$, of the past action, a_{t-1}, the latest observation, o_t, the predictions from the previous time step, $\mathbf{y}_{t-1}$, and a modifiable parameter vector, $\boldsymbol{\theta}_t$.

$$\mathbf{y}_t = \mathbf{u}(\mathbf{y}_{t-1}, a_{t-1}, o_t, \boldsymbol{\theta}_t). \tag{4.3}$$

Generally, $\mathbf{u}$ is a function which applies an operator $\boldsymbol{\sigma}$ to the linear combination of the parameter vector, $\boldsymbol{\theta}_t$ and the feature vector, $\boldsymbol{\phi}_t$:

$$\mathbf{y}_t = \boldsymbol{\sigma}(\boldsymbol{\theta}_t^\top \boldsymbol{\phi}_t), \tag{4.4}$$

where $\boldsymbol{\sigma}$ has been either the vector form of the identity function or the S-shaped logistic function $\sigma(s) = \frac{1}{1+e^{-s}}$ in existing TD network literature (Sutton & Tanner, 2004; Tanner

[2]Due to issues with timing, the target is a function of the observation, o_t and $\tilde{\mathbf{y}}_{t+1}$ (formally defined in Equation 4.20). For clarity, the intermediate predictions can be thought of as $\mathbf{y}_{t+1}$.

& Sutton, 2005a; Tanner & Sutton, 2005b; Tanner, 2005). The feature vector at time step t, $\boldsymbol{\phi}_t$, is constructed by a function $\boldsymbol{\phi}$ that maps the predictions from the previous time-step, the last action, and the current observation to an m-dimensional set of features ($\boldsymbol{\phi} : \mathcal{R}^n \times \mathcal{A} \times \mathcal{O} \rightarrow \mathcal{R}^m$):

$$\boldsymbol{\phi}_t = \boldsymbol{\phi}(\mathbf{y}_{t-1}, a_{t-1}, o_t). \tag{4.5}$$

A gradient descent learning rule is used to update the weights in order to minimize the prediction error ($z_t^i - y_t^i$):

$$\theta_{t+1}^{ij} = \theta_t^{ij} + \alpha(z_t^i - y_t^i)c_t^i \frac{\partial y_t^i}{\partial \theta_t^{ij}}, \tag{4.6}$$

where α is a positive step-size parameter.

All predictions in the framework described in this section are conditioned on actions that span a single time-step. Predictions about events k steps in the future can be made by chaining together k predictive nodes (see Section 2.2); however, it is not possible to model sequences of arbitrary length.

4.2 Options

The option framework (Sutton, Precup, & Singh, 1999) is an approach to representing temporally abstract knowledge in reinforcement learning algorithms. Options—a generalization of actions—consists of three components: an initiation set, a policy, and a termination condition. The initiation set, $\mathcal{I} \subset \mathcal{S}$, indicates the states from which the option can begin. The policy, $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$, specifies the probability of selecting a given action in a given state. The termination condition, $\beta(s) : \mathcal{S} \rightarrow [0, 1]$, is the probability that the option will terminate in any given state. The definition of options presented here is for MDPs, but can be generalized to partially observable environments by defining the three option components over histories rather than states.

Options are used in reinforcement learning to predict expected reward and expected state upon termination. Precup et. al demonstrate that a reinforcement-learning agent can interrupt an option during execution if a different action or option would result in higher reward. Non-terminating executions of an option can still be used to improve the predictions that it makes. In this chapter, options are always followed until termination; in Chapter 5, learning from incomplete trajectories is incorporated into our algorithm.

4.3 Option-conditional TD (OTD) Networks

Options are integrated into the temporal-difference network framework as a means for representing temporally abstract predictions. In the OTD-network framework introduced in this thesis, the answer network is modified to include termination conditions $\boldsymbol{\beta}$ and an $n \times m$ eligibility trace matrix $\mathbf{E}$.

As with TD networks (Section 4.1), predictions in an OTD network are computed as specified in Equation 4.3 and the feature vector is constructed according to Equation 4.5. Targets are defined as in Equation 4.2, but conditions are now based on options rather than on actions. The condition at time t, $c_t^i = c^i(a_t, \mathbf{y_t})$, is a binary variable that indicates whether an option is being followed. Learning is conducted on-policy: $c_t^i = 1$ from $t = l, \ldots, T$, where l is the time step at which option i (the option corresponding to prediction y^i) is initiated and T is the time step at which the option terminates (according to the termination condition β^i). If option i is not being followed, then $c_t^i = 0$. With on-policy learning, updates are only permitted when an option is followed until termination. If option i has been initiated, but the agent ceases to choose actions from the option's policy, π^i, then the agent is said to have *diverged*, and any weights updated by the agent since option initiation are reverted to their pre-initation values.

A termination function $\beta^i : \mathcal{O} \times \mathcal{R}^n \to [0, 1]$ is defined as $\beta_t^i = \beta^i(o_t, \mathbf{y}_{t-1})$. If $\beta_t^i = 1$ the option terminates at time t. It is also possible that $0 < \beta_t^i < 1$, indicating that the option randomly terminates with probability β_t^i on time step t. The value of β_t^i has a part in determining the prediction error as it trades responsibility between node i's target, z_t^i and the node's own prediction on the next time step $\tilde{y}_t^i$:

$$\delta_t^i = \beta_{t+1}^i z_{t+1}^i + (1 - \beta_{t+1}^i)\tilde{y}_{t+1}^i - y_t^i. \tag{4.7}$$

An eligibility trace matrix, $\mathbf{E}_t$, keeps track of active inputs throughout the course of an option's execution. Individual components of the matrix, e_t^{ij}, are updated according to:

$$e_t^{ij} = \lambda(1 - \beta_t^i)e_{t-1}^{ij} + \frac{\partial y_t^i}{\partial \theta_t^{ij}}. \tag{4.8}$$

On each time step traces are decayed by a factor of $0 \leq \lambda \leq 1$. When $\beta_t^i = 1$ (when option i terminates), the previous traces disappear, thus immediately beginning a new trace.

Elements of the $n \times m$ weight matrix $\boldsymbol{\theta}_t$ are updated according to:

$$\theta_{t+1}^{ij} = \theta_t^{ij} + \alpha\delta_t^i c_t^i e_t^{ij} \tag{4.9}$$

where α is a positive step-size parameter. As with TD networks, the weights are updated in the direction of the prediction error gradient. Note that if an option policy is not being followed ($c_t^i = 0$), then the weights do not change.

The order of computation is:

$$\mathbf{y}_t \; a_t \; \mathbf{c}_t \; \mathbf{E}_t \; o_{t+1} \; \boldsymbol{\phi}_{t+1} \; \tilde{\mathbf{y}}_{t+1} \; \boldsymbol{\beta}_{t+1} \; \mathbf{z}_{t+1} \; \boldsymbol{\delta}_t \; \boldsymbol{\theta}_{t+1} \; \mathbf{y}_{t+1}. \tag{4.10}$$

Pseudocode can be found in Algorithm 1.

Algorithm 1 The on-policy OTD network learning algorithm.

Initialize $\mathbf{y}_0, \mathbf{E}_0, \boldsymbol{\theta}_0, \boldsymbol{\beta}_0$
for $t = 1, 2, \ldots$
  Take action a_t; receive observation o_{t+1}
  Compute conditions: $\mathbf{c}_t = \mathbf{c}(a_t, \mathbf{y_t})$
  Update trace matrix: $\mathbf{E}_t = \mathbf{c}_t(\lambda(1 - \boldsymbol{\beta}_t)\mathbf{E}_{t-1} + \nabla_{\boldsymbol{\theta}_t}\mathbf{y}_t)$
  Construct feature vector: $\boldsymbol{\phi}_{t+1} = \boldsymbol{\phi}(\mathbf{y}_t, a_t, o_{t+1})$
  Compute interim predictions: $\tilde{\mathbf{y}}_{t+1} = \mathbf{u}(\boldsymbol{\phi}_{t+1}, \boldsymbol{\theta}_t)$
  Check for termination: $\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}(o_{t+1}, \mathbf{y}_t)$
  Update target values: $\mathbf{z}_t = \mathbf{z}(o_{t+1}, \tilde{\mathbf{y}}_{t+1})$
  Compute error: $\boldsymbol{\delta}_t = \boldsymbol{\beta}_{t+1}\mathbf{z}_{t+1} + (1 - \boldsymbol{\beta}_{t+1})\tilde{\mathbf{y}}_{t+1} - \mathbf{y}_t$
    (multiplications are component-wise)
  Update weights: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\boldsymbol{\delta}_t\mathbf{E}_t$
  Update predictions: $\mathbf{y}_{t+1} = \mathbf{u}(\boldsymbol{\phi}_{t+1}, \boldsymbol{\theta}_{t+1})$
end for

4.4 Algorithm Derivation

The equations in the previous section were obtained by deriving a backwards view algorithm from a forward view algorithm that is defined in this section.

An option model—the expected value of some target quantity upon an option's termination—is being learned for each prediction. The quantity being predicted upon termination, z_T, is called the *outcome* of the option. As with TD-networks, the expected value of the outcome is generally approximated as a linear combination of the features $\boldsymbol{\phi}_t$:

$$y_t = E[z_T|\pi, \beta] \approx \theta_t^\top \phi_t. \tag{4.11}$$

The option is assumed to terminate in finite time according to β at time T.

Note that in this section the superscript i is dropped and the algorithm derivation is conducted with respect to a single prediction, though the derivation generalizes to multiple predictions.

4.4.1 The Forward View

The forward view of an algorithm is a theoretical entity, relying on an oracle to provide the value of z_{t+n}, the target n steps in the future. Let $Z_t^{(n)}$ be the n-step outcome, defined (recursively) as:

$$Z_t^{(n)} = \beta_{t+1} z_{t+1} + (1 - \beta_{t+1}) Z_{t+1}^{(n-1)}, \tag{4.12}$$

where the base case is $Z_t^{(0)} = y_t$. This equation says that for an outcome $Z_t^{(n)}$, if the option terminates at time $t+1$, the quantity z_{t+1} is used as a target, but if the option does not terminate at time $t+1$, the latest prediction, y_t, is used as a target. The equation can be better understood by unrolling the recursion for small values of n.

$$\begin{aligned}
Z_t^{(1)} &= \beta_{t+1} z_{t+1} + (1 - \beta_{t+1}) y_{t+1} \\
Z_t^{(2)} &= \beta_{t+1} z_{t+1} + (1 - \beta_{t+1})(\beta_{t+2} z_{t+2} + (1 - \beta_{t+2}) y_{t+2}) \\
Z_t^{(3)} &= \beta_{t+1} z_{t+1} + (1 - \beta_{t+1})(\beta_{t+2} z_{t+2} + (1 - \beta_{t+2})(\beta_{t+3} z_{t+3} + (1 - \beta_{t+3}) y_{t+3}) \\
&\vdots
\end{aligned}$$

The forward view equations have a clear interpretation in the case where β_t is binary. For the 1-step outcome ($Z_t^{(1)}$), the value β_{t+1} determines whether the outcome is the target z_{t+1} or the latest prediction y_{t+1}. For the 2-step outcome ($Z_t^{(2)}$), if the option terminates at time $t+1$, then the target at this time step, z_{t+1}, is used as the outcome. If the option does not terminate at time $t+1$, then the outcome is a value from time-step $t+2$. β_{t+2} now determines whether the target z_{t+2} or the latest prediction y_{t+2} is used as an outcome. A similar pattern is followed for the 3-step outcome (graphically represented in Figure 4.1).

A λ-outcome combines the n-step outcomes:

$$Z_t^{\lambda} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} Z_t^{(n)}. \tag{4.13}$$

The λ-return is an exponentially-weighted average of all future n-step returns, putting more weight on lower values of n, and consequently on outcomes closer to the current time step t.

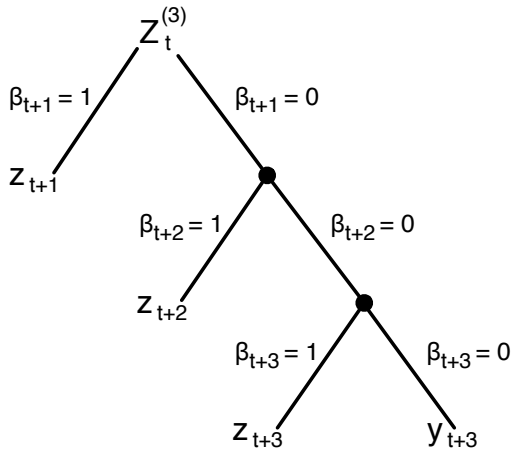


Figure 4.1: A graphical representation of the n-step outcome for $n = 3$. The possible outcomes are z_{t+1}, z_{t+2}, z_{t+3}, or y_{t+3} depending on the values of β_{t+1}, β_{t+2}, and β_{t+3}.

Finally, the weight update in the direction of the error gradient is

$$\Delta\theta_{t+1} = \alpha(Z_t^\lambda - y_t)\nabla_\theta y_t. \tag{4.14}$$

where α is a positive step-size parameter. The sum of weight updates over the course of an option's execution is thus:

$$\Delta\theta = \sum_{t=0}^{T} \alpha(Z_t^\lambda - y_t)\nabla_\theta y_t. \tag{4.15}$$

4.4.2 Forward and Backward View Equivalence

The preceding forward definition of temporal-difference networks with options is used to derive an algorithm with incremental updates (the backward-view algorithm). It is convenient to express the error term $(Z_t^\lambda - y_t)$ from the forward view in a different form:

$$\begin{aligned}
Z_t^\lambda - y_t = &- y_t \\
&+ (1-\lambda)\lambda^0\big(\beta_{t+1}z_{t+1} + (1-\beta_{t+1})y_{t+1}\big) \\
&+ (1-\lambda)\lambda^1\big(\beta_{t+1}z_{t+1} + (1-\beta_{t+1})[\beta_{t+2}z_{t+2} + (1-\beta_{t+2})y_{t+2}]\big) \\
&+ (1-\lambda)\lambda^2\big(\beta_{t+1}z_{t+1} + (1-\beta_{t+1})[\beta_{t+2}z_{t+2} + (1-\beta_{t+2}) \\
&\qquad\qquad (\beta_{t+3}z_{t+3} + (1-\beta_{t+3})y_{t+3})]\big) \\
&+ \cdots
\end{aligned}$$

$$
\begin{aligned}
&= -y_t \\
&\quad + \lambda^0\big(\beta_{t+1}z_{t+1} + (1-\beta_{t+1})y_{t+1} - \lambda[\beta_{t+1}z_{t+1} + (1-\beta_{t+1})y_{t+1}]\big) \\
&\quad + \lambda^1\big(\beta_{t+1}z_{t+1} + (1-\beta_{t+1})[\beta_{t+2}z_{t+2} + (1-\beta_{t+2})y_{t+2}] \\
&\qquad\quad - \lambda[\beta_{t+1}z_{t+1} + (1-\beta_{t+1})(\beta_{t+2}z_{t+2} + (1-\beta_{t+2})y_{t+2})]\big) \\
&\quad + \lambda^2\big(\beta_{t+1}z_{t+1} + (1-\beta_{t+1})\big[\beta_{t+2}z_{t+2} + (1-\beta_{t+2}) - \\
&\qquad\qquad\qquad (\beta_{t+3}z_{t+3} + (1-\beta_{t+3})y_{t+3})\big] \\
&\qquad\quad - \lambda\big[\beta_{t+1}z_{t+1} + (1-\beta_{t+1})(\beta_{t+2}z_{t+2} + (1-\beta_{t+2}) \\
&\qquad\qquad\qquad [\beta_{t+3}z_{t+3} + (1-\beta_{t+3})y_{t+3}])\big]\big) \\
&\quad + \cdots \\
&= \quad \lambda^0(\beta_{t+1}z_{t+1} + (1-\beta_{t+1})y_{t+1} - y_t) \\
&\quad + \lambda^1(\beta_{t+1}z_{t+1} + (1-\beta_{t+1})[\beta_{t+2}z_{t+2} + (1-\beta_{t+2})y_{t+2}] \\
&\qquad\qquad\qquad - [\beta_{t+1}z_{t+1} + (1-\beta_{t+1})y_{t+1}]) \\
&\quad + \lambda^2\big(\beta_{t+1}z_{t+1} + (1-\beta_{t+1})[\beta_{t+2}z_{t+2} + (1-\beta_{t+2}) \\
&\qquad\qquad\qquad (\beta_{t+3}z_{t+3} + (1-\beta_{t+3})y_{t+3})] \\
&\qquad\quad - [\beta_{t+1}z_{t+1} + (1-\beta_{t+1})(\beta_{t+2}z_{t+2} + (1-\beta_{t+2})y_{t+2})]\big) \\
&\quad + \cdots \\
&= \lambda^0(Z_t^{(1)} - y_t) + \\
&\quad \lambda^1(Z_{t+1}^{(1)} - y_{t+1})(1-\beta_{t+1}) + \\
&\quad \lambda^2(Z_{t+2}^{(1)} - y_{t+2})(1-\beta_{t+1})(1-\beta_{t+2}) \\
&\quad + \cdots \\
&= \sum_{i=t}^{\infty} \lambda^{i-t}\delta_i \prod_{j=t+1}^{i}(1-\beta_j) \qquad (4.16)
\end{aligned}
$$

where

$$
\begin{aligned}
\delta_i &= (Z_i^{(1)} - y_i) \\
&= \beta_{i+1}z_{i+1} + (1-\beta_{i+1}y_{i+1}) - y_i \qquad (4.17)
\end{aligned}
$$

The forward and the backward views are equivalent because both views have the same sum of updates over the course of an option's execution (shown next). Only considered are updates from option initiation at time-step 0 until option termination at time T. Equa-

tion 4.16 is therefore a finite sum:

$$\sum_{i=t}^{\infty} \lambda^{i-t}\delta_i \prod_{j=t+1}^{i}(1-\beta_j) = \sum_{i=t}^{T} \lambda^{i-t}\delta_i \prod_{j=t+1}^{i}(1-\beta_j).$$

Elements in the summation for $i > T$ are not considered because the post-termination value for the product $\prod_{j=t+1}^{i}(1-\beta_j)$ is 0.

In the sum of weight updates (Equation 4.15), the error term $(Z_t^\lambda - y_t)$ can be replaced by Equation 4.16 and the summation property

$$\sum_{i=0}^{N}\sum_{j=i}^{N} a_{ji} = \sum_{i=0}^{N}\sum_{j=0}^{i} a_{ij}$$

is used to re-express the sum of updates:

$$\begin{aligned}
\sum_{t=0}^{T} \alpha(Z_t^\lambda - y_t)\nabla_\theta y_t &= \sum_{t=0}^{T} \alpha\Big(\sum_{i=t}^{T} \lambda^{i-t}\delta_i \prod_{j=t+1}^{i}(1-\beta_j)\Big)\nabla_\theta y_t \\
&= \sum_{t=0}^{T} \alpha\delta_t \sum_{i=0}^{t} \lambda^{t-i}\nabla_\theta y_i \prod_{j=i+1}^{t}(1-\beta_j) \\
&= \sum_{t=0}^{T} \alpha\delta_t e_t \qquad (4.18)
\end{aligned}$$

where $e_t = \sum_{i=0}^{t} \lambda^{t-i}\nabla_\theta y_i \prod_{j=i+1}^{t}(1-\beta_j)$.

The condition variable c_t does not appear in the derivation because $c_t = 1$ for $0 \leq t \leq T$ during on-policy learning and therefore,

$$\sum_{t=0}^{T} \alpha\delta_t e_t = \sum_{t=0}^{T} \alpha\delta_t c_t e_t$$

The next step in deriving a backwards view for the OTD network algorithm is to define e_t incrementally. Equation 4.8 can be shown to be correct via induction.

Theorem 4.4.1

$$e_t = \sum_{i=0}^{t} \lambda^{t-i}\nabla_\theta y_i \prod_{j=i+1}^{t}(1-\beta_j) = \lambda(1-\beta_t)e_{t-1} + \nabla_\theta y_t \qquad (4.19)$$

$$e_0 = \nabla_\theta y_0$$

Proof The bases case are equivalent by definition:

$$e_0 = \sum_{i=0}^{0} \lambda^{0-i}\nabla_\theta y_i \prod_{j=i+1}^{0}(1-\beta_j) = \nabla_\theta y_0$$

Next, assuming that Equation 4.19 is true for e_t:

$$
\begin{aligned}
e_{t+1} &= \lambda(1-\beta_{t+1})e_t + \nabla_\theta y_{t+1} \\
&= \lambda(1-\beta_{t+1})\left(\sum_{i=0}^{t}\lambda^{t-i}\nabla_\theta y_i \prod_{j=i+1}^{t}(1-\beta_j)\right) + \nabla_\theta y_{t+1} && \text{Equation 4.19} \\
&= \sum_{i=0}^{t}\left(\lambda^{(t+1)-i}\nabla_\theta y_i \prod_{j=i+1}^{t+1}(1-\beta_j)\right) + \nabla_\theta y_{t+1} && c\sum_{i}^{n} a = \sum_{i}^{n} ca \\
&= \sum_{i=0}^{t+1}\lambda^{t-i}\nabla_\theta y_i \prod_{j=i+1}^{t}(1-\beta_j) && \lambda^{(t+1)-(t+1)}\nabla_\theta y_{t+1}\prod_{j=(t+1)+1}^{t+1}(1-\beta_j) = \nabla_\theta y_{t+1}
\end{aligned}
$$

∎

Finally, the quantity $\tilde{y}_{t+1}$ must be defined. The prediction on the next time-step, y_{t+1}, is not yet available for computing δ_t so $\tilde{y}_{t+1}$ serves as an approximation:

$$\tilde{y}_{t+1} = \boldsymbol{\theta}_t^\top \boldsymbol{\phi}_{t+1}. \tag{4.20}$$

Thus the TD error is computed as:

$$\delta_t = \beta_{t+1} z_{t+1} + (1-\beta_{t+1})\tilde{y}_{t+1} - y_t. \tag{4.21}$$

4.5 OTD Network Experiments

This section begins with the presentation of an example grid world and the corresponding OTD network that will be used as a running example throughout this thesis. The error metric used throughout the rest of this thesis is also described in this section. In addition, results of the on-policy OTD network algorithm in the example grid world are presented.

4.5.1 The Environment

The grid-world in Figure 4.2 will serve as a running example for the rest of this thesis. The agent can occupy any of the 36 white grid cells and can be in any of the four cardinal directions (North, South, East, or West)—a total of 144 environmental states. However, the agent does not directly observe its environmental state. Instead, it observes a six-element bit vector, where each bit corresponds to a color (blue, green, orange, red, yellow, and white). The color of the that the agent is facing determines which bit is set to 1; all other bits will have a value of 0. As described in Section 3.2, the agent has three actions available: step forward (F), rotate 90° right (R), and rotate 90° left (L).

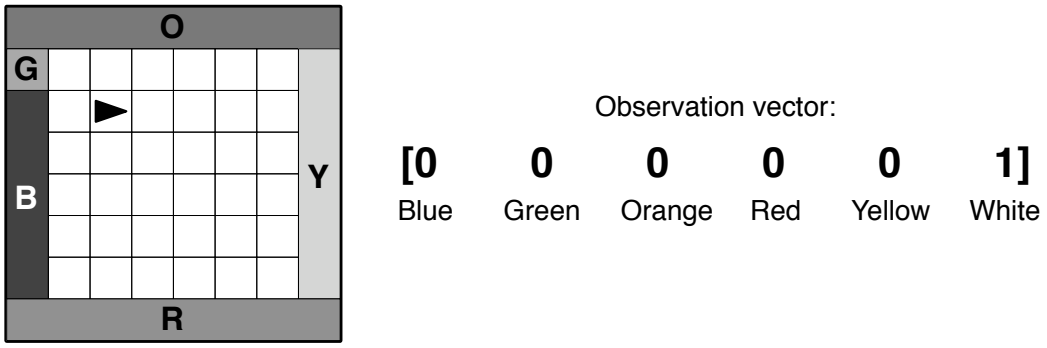


Figure 4.2: The grid-world environment (left) is used for discussion and experiments throughout the rest of this thesis. In each of the 36 white grid cells the agent can be in one of four orientations (facing North, South, East, West). There are therefore 144 environmental states. The agent (denoted by the triangle) can step forward (F), rotate 90° right (R), or rotate 90° left (L). The agent receives a 6-element bit vector (right) as an observation. The bit corresponding to the color of the grid cell (blue (B), green (G), orange (O), yellow (Y), red (R), or white (W)) that the agent is immediately facing will have a value of 1, while all other bits will be set to 0.

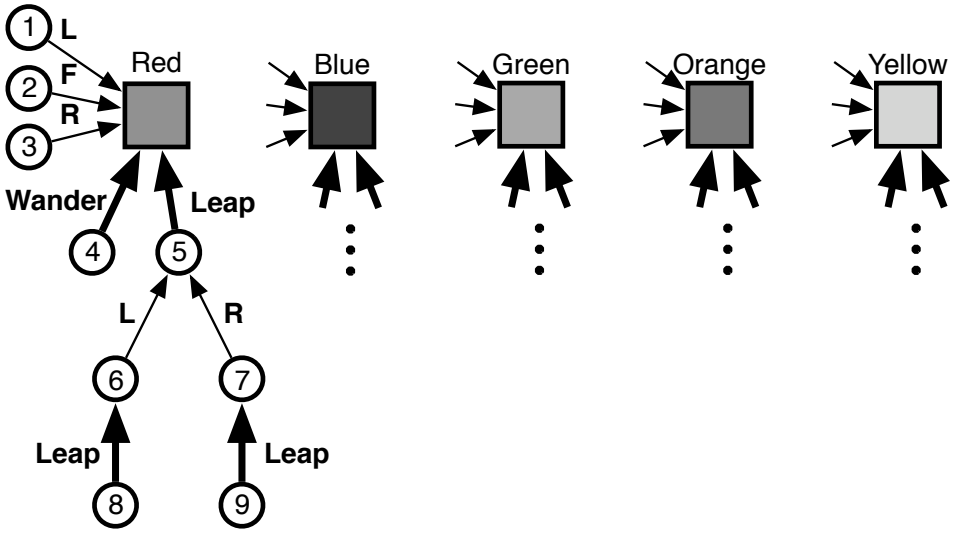


Figure 4.3: An illustration of the question network used in the experiments in this section. The nine-node structure is repeated five times, one for each non-white color (Red, Blue, Green, Orange, Yellow). The predictions are for the outcomes of: 1) Rotate Left, 2) Step Forward, 3) Rotate Right, 4) **Wander**, 5) **Leap**, 6) Rotate Left then **Leap**, 7) Rotate Right then **Leap**, 8) **Leap**, Rotate Left, then **Leap**, and 9) **Leap**, Rotate Right, then **Leap**. The **Wander** and **Leap** options are described in detail in the text. All 45 nodes in the TD network are interconnected by the answer network.

4.5.2 The Temporal-difference Network

The temporal-difference network used in the experiments is shown in Figure 4.3. While there is a combination of simple actions and options as conditions, for simplicity all nodes are considered option-conditional because options are a generalization of simple actions. The actions F, R, and L can be expressed as options that:

- can be initiated in any state ($\mathcal{I} = \mathcal{S}$);
- have a policy that always chooses the associated action ($p(\cdot, F) = 1$, $p(\cdot, R) = 1$, and $p(\cdot, L) = 1$);
- always terminate after a single time-step ($\beta(s) = 1.0, \forall s \in \mathcal{S}$).

Nodes 1, 2, and 3 represent single-step predictions about step forward, rotate right, and rotate left. These nodes correspond to the three questions: "If I step forward, will the red observation bit be 1?", "If I rotate right, will the red observation bit be 1?", and "If I rotate left, will the red observation bit be 1?" Node 4 predicts the outcome of the **Wander** option, whose policy is to choose all actions with equal probability and whose termination condition is $\beta(o_t = \text{white}) = 0.5$ and $\beta(o_t \neq \text{white}) = 1.0$ (50% chance of termination if the agent is facing a white grid cell an 100% chance of termination if the agent is facing a colored grid cell). Node 5 predicts the outcome of the **Leap** option, whose policy is to always take the step forward action ($p(\cdot, F) = 1$) and whose termination condition is $\beta(o_t = \text{white}) = 0.0$ and $\beta(o_t \neq \text{white}) = 1.0$. This node asks the question: "If I step forward until I see a wall, will the wall be red?" Nodes 6 and 7 are compositions of the **Leap** option with rotate left and rotate right. These nodes predict the value of Node 5 if the agent were to rotate right or rotate left. Extensively, Nodes 6 and 7 predict the value of the observation bit if the agent were to rotate left or rotate right then follow the **Leap** option until termination, thus asking the question "If I rotate right(left) then follow the **Leap** option until termination, will the red observation bit be 1?" Similarly, Nodes 8 and 9 make predictions about other predictions. They predict the values of Nodes 6 and 7 if the **Leap** option were to be followed until termination. The extensive question asked by these nodes is: "If I follow the **Leap** option until termination, rotate right(left) then, again, follow the **Leap** option until termination, will the red observation bit be 1?"

As suggested by Figure 4.3, the nine-node structure is repeated five times (once for each non-white bit). In total, there are 45 predictions being made on each time step.

The agent constructs a 156-element feature vector for use as a representation. The feature vector $\boldsymbol{\phi}_t$, is constructed from the agent's last predictions, current observations and past action. This feature vector has 156 elements, divided into three groups of 52-elements each—one group for each of the three actions. The first element of a 52-element group is a bias term, which is always 1. The next six elements are the agent's 6-bit observation (o_t). The remaining 45 elements are the 45 TD network predictions from the previous time step ($\mathbf{y}_{t-1}$). If the action taken (a_t) was the step-forward action, then the values for the first 52-element section are filled in as described above, while the other 104 elements are assigned values of 0. If the rotate-right action was taken, then only the middle 52 elements (elements 53-104) are filled in, and if the rotate-left action was taken, then only the last 52 elements (elements 105-156) are filled in.

Predictions are computed as the dot product of the parameter vector $\boldsymbol{\theta}_t$ and the feature vector $\boldsymbol{\phi}_t$, subject to some function $\boldsymbol{\sigma}$ (Equation 4.4). In all the experiments throughout this thesis, $\boldsymbol{\sigma}$ is a bounded identity function. For each node i:

$$\sigma^i(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } 0 \leq x \leq 1 \\ 1 & \text{if } x > 0 \end{cases}$$

The values of θ_0, E_0, and y_0 were always initialized to 0.

4.5.3 Error Metric

The quality of the predictions made by the OTD network was measured by comparing the predictions to values generated by an oracle. At each environmental state, each node's sequence of options was simulated in order to determine the correct prediction. The predictions corresponding to Nodes 1-3 and 5-9 (see Figure 4.3) were determined by following each sequence once (because the environment is deterministic). However, for Node 4, ten thousand **Wander** trajectories were generated and the average outcome was used as the oracle value.

On each time step t, the squared error was calculated for each node i:

$$error^2(i,t) = (y(i,t) - y^*(i,t))^2, \tag{4.22}$$

where $y^*(i,t)$ is the oracle value[3]. The root mean square error of each node ($RMSE(i)$) was recorded every N steps:

$$RMSE(i) = \sqrt{\frac{\sum_{j=0}^{N} error^2(i,t+j)}{N}}. \tag{4.23}$$

[3]The notation has been altered slightly for the purposes of clarity. Previously in this chapter y_t^i was used to denote the prediction of node i at time t.

Network error is defined as:

$$network\_error = \frac{\sum_{i=0}^{45} RMSE(i)}{45}, \tag{4.24}$$

the average error of the network's 45 predictions.

4.5.4 Parameter Study

We tested the OTD network algorithm for all combinations of $\alpha = \{0.01,\ 0.05,\ 0.1\}$ and $\lambda = \{0,\ 0.25,\ 0.5,\ 0.75,\ 1\}$. All learning was performed on-policy; only nodes whose policies matched the current behavior were updated. Simple actions were expressed as options as described in Section 4.5.1 and the agent could choose from five options: step forward, rotate right, rotate left, **Leap**, and **Wander**. When an option terminated (options were always followed until termination), the agent chose a new option. Options were randomly chosen according to the following distribution:

- Step forward: % 50
- Rotate right: % 20
- Rotate left: % 20
- **Leap**: % 5
- **Wander**: % 5

Results of the experiments are shown in Figure 4.4. The curves pictured in the graphs are network errors averaged over 10 runs of 250,000 steps ($N = 10,000$) for each parameter setting. In all experiments the speed of learning improved as λ approached 1. At the end of training, the average network error was similar for $\alpha = 0.01$ and $\alpha = 0.05$, both of which were better than the average network error for $\alpha = 0.1$. The best combination of learning rate and post-training network error was $\alpha = 0.05$ and $\lambda = 1.0$.

When training continued beyond the 250,000 steps, the network error continued to decrease slowly over time. However, it is important to note that the average network error will never reach zero because certain predictions cannot be made perfectly.

4.5.5 Individual Node Error

For some nodes, it is possible to completely eliminate prediction error. An example of this is the prediction of Node 5 in Figure 4.3, the prediction of whether the red observation bit

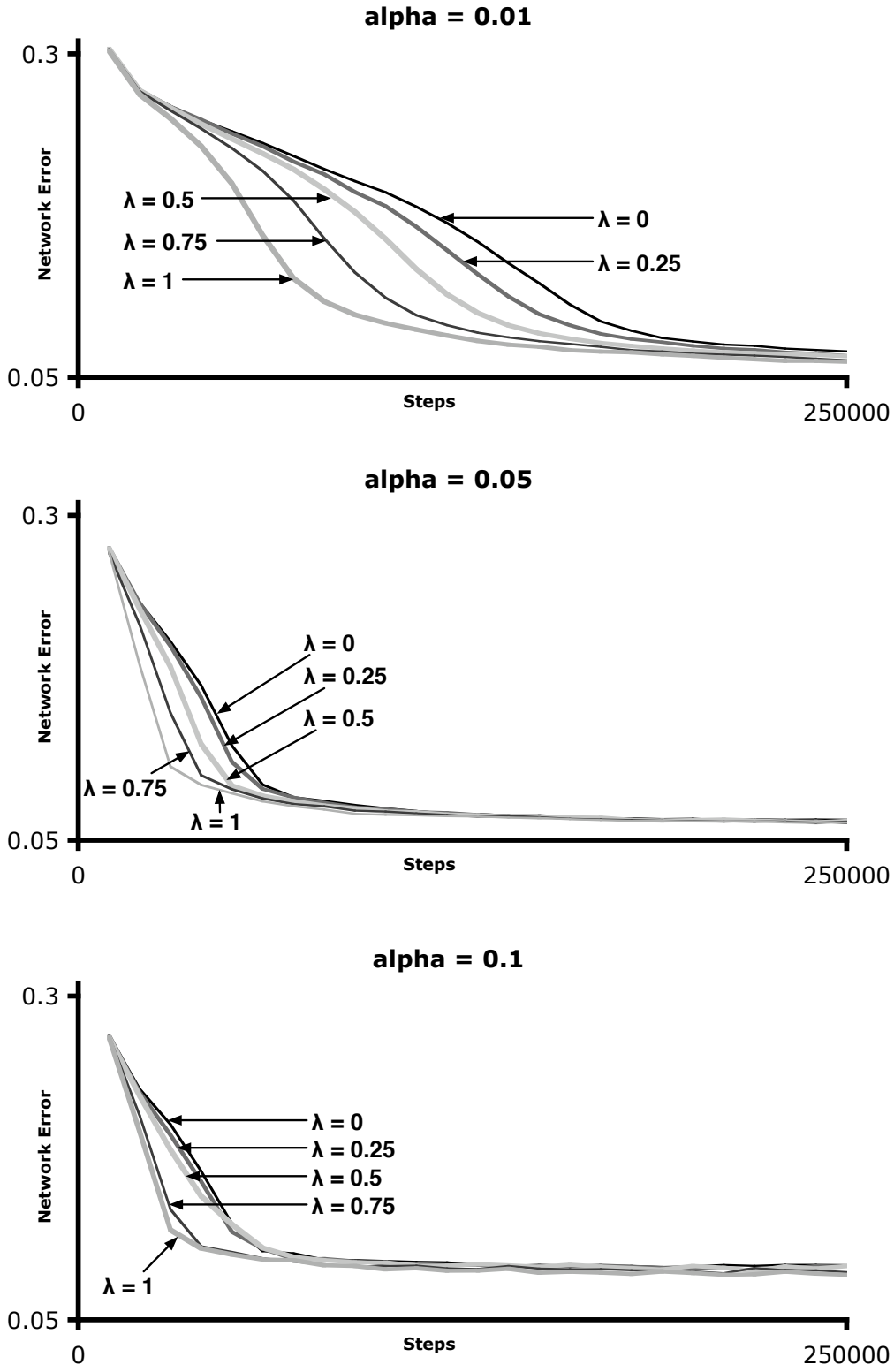


Figure 4.4: Experimental results with the on-policy OTD network algorithm for all combinations of $\alpha = \{0.01, 0.05, 0.1\}$ and $\lambda = \{0, 0.25, 0.5, 0.75, 1\}$. The curves are network errors averaged over 10 runs of 250,000 steps for each parameter setting. See Section 4.5.3 for a description of how average network error was calculated.

will be 1 if the **Leap** option is taken until termination. The result of learning is that the agent maintains an internal concept of direction and thus if the agent is facing the direction of the red wall, then the prediction for Node 5 will be 1. Node 5's prediction will be 0 if the agent is facing any other direction. These values match the oracle values and prediction error quickly drops to 0 (see Figure 4.5).

However, certain predictions cannot be made perfectly. When facing west, the **Leap** option could result in either observing the blue bit or observing the green bit depending on which row the agent is in. Despite the fact that the agent can maintain an internal representation of direction, none of the predictions can help distinguish which row or column it is in. Instead of making a binary prediction about the outcome of the **Leap** option, the agent predicts an intermediate value between 0 and 1—the value corresponding to the probability that either the blue bit or the green bit will be observed.

Prediction errors of the **Leap** nodes is the subject of Figure 4.5. These curves graph node errors for the nodes predicting red, blue, and green observations bits conditioned on the action sequences shown. The curves are averages over 30 runs of 100,000 steps with the parameter settings $\alpha = 0.05$ and $\lambda = 1$.

The prediction error for the **Leap** nodes quickly drops to 0 for the red observation bit, but not for the blue and green observation bits. Though these nodes show gradual improvement over the course of training, prediction error remains. The prediction errors for the orange and yellow observation bits (not pictured) follow a curve very similar to the red's error curve.

As a result of the prediction error in the blue and green **Leap** nodes, Nodes 6 and 7 of Figure 4.3 also err in their predictions of blue and green. The error is propagated from Node 5 (**Leap**) to Nodes 6 and 7 (R-**Leap** and L-**Leap**) because Nodes 6 and 7 make predictions about the value of Node 5. If the prediction of Node 5 has an error, Nodes 6 and 7 use this erroneous value as a target. The individual node errors for the predictions of L-**Leap** and R-**Leap** are displayed in Figure 4.5. Again, the prediction error for the red observation bit quickly drops to 0 while the predictions for blue and green contain error.

There is a noticeable difference between the prediction errors for L-**Leap** and R-**Leap**. The prediction errors for the blue and green observations are much lower for L-**Leap**. This is likely due to the placement of the green grid cell in the environment (cf. Figure 4.2). When facing North, the agent learns that the observation of the orange bit informs the agent that the sequence L-**Leap** will lead to an observation of green. When facing South,

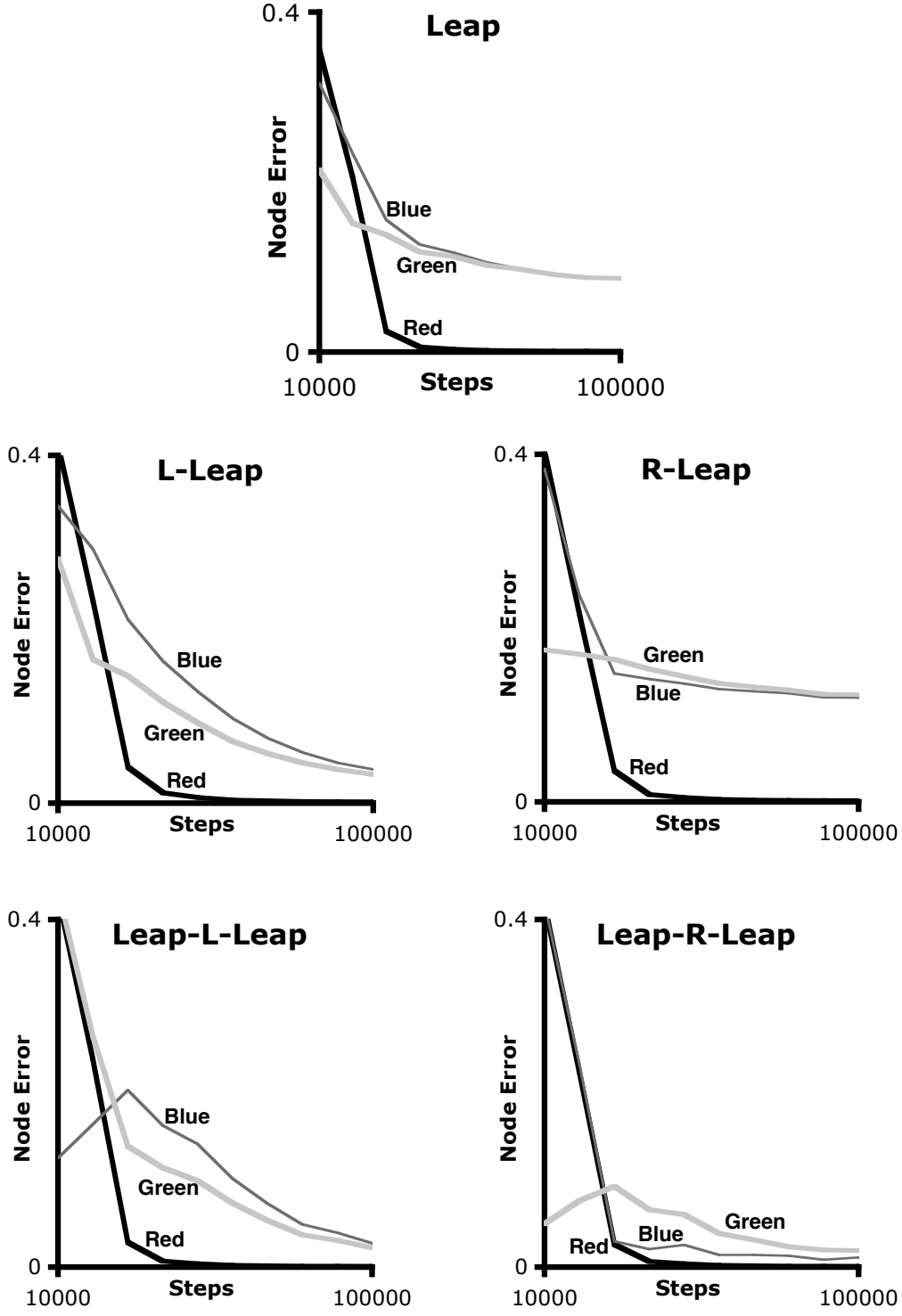


Figure 4.5: Individual errors for the **Leap** nodes averaged over 30 runs of 100,000 steps ($N = 10,000$). The values were learned with the parameter settings, $\alpha = 0.05$ and $\lambda = 1$. The curves are for the predictions about the red, blue, and green observation bits. The predictions related to the red observation bit can be made perfectly while the probabilistic predictions about blue and green improve, but do not reach 0.

neither the observations nor the predictions in the network provide provide the information necessary to distinguish whether R-**Leap** would lead to observing green or blue.

The error is lower for the predictions of the sequences **Leap**-L-**Leap** and **Leap**-R-**Leap** (Nodes 8 and 9 of Figure 4.3) then for the predictions for L-**Leap** and R-**Leap**. Nodes 8 and 9's sequence of options effectively localizes the agent in a specific corner and orientation, removing any ambiguity about the agent's location. The first **Leap** option takes the agent to the wall it is facing, the agent then rotates either right or left, then the second **Leap** option takes the agent into a corner. The prediction error for these nodes is found in Figure 4.5. Though the graphs in this figure stop after 100,000 steps, if the graph were to be extend further, the prediction error would continue to approach 0.

The agent cannot always make the correct single-step predictions (Nodes 1, 2, and 3). For instance, when predicting the outcome of the step-forward action, the agent has an internal concept of direction from the predictions made by the **Leap** nodes, but none of the predictions indicate the agent's distance from the wall. From the middle of the grid world and facing the red wall, the agent's prediction that the step-forward action will result in an observation of red is between 0 and 1. With each subsequent step forward, the agent continues to predict that with some probability red will be observed. Eventually the agent collides with the colored grid cell and at that point it predicts red with complete certainty. The agent cannot make perfect predictions, because in the error measurement, the oracle value for F will be a binary value. Any prediction between 0 and 1 will result in prediction error. Therefore, none of the predictions for the simple actions (Nodes 1, 2, and 3 of Figure 4.3 for all different colors) can be made perfectly at all times. Predicting without error is only possible in specific situations. For example, when the agent is immediately facing a colored grid cell, the agent correctly predicts that the step-forward action will result in an observation of the same color. The prediction errors for the red, blue, and green observation bits for the F, L, and R actions are shown in Figure 4.6. These graphs show a marginal improvement in the quality of the predictions early in training, but very little change thereafter.

There is also substantial error in the predictions of the outcome of the **Wander** option. The presence of error may be related to the issue described for the R-**Leap** and the L-**Leap** nodes: the predictions do not provide sufficient information for the agent to determine its exact position in the environment. The agent's predictions are compared to oracle values which were computed for each environmental state. The agent cannot distinguish its position

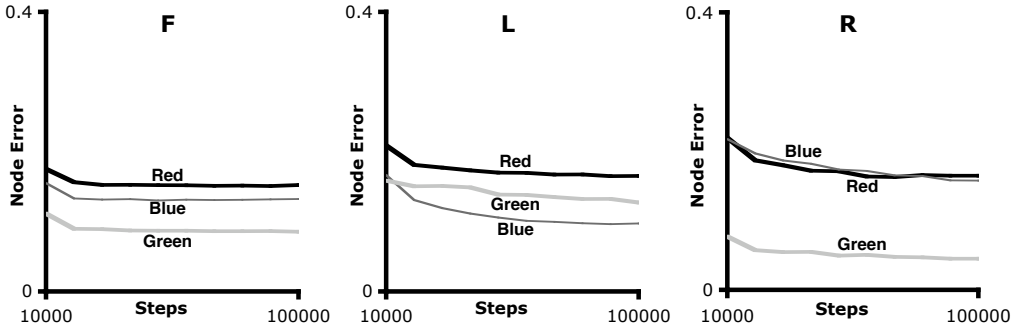


Figure 4.6: Individual node errors for the F,L, and R nodes averaged over 30 runs of 100,000 steps. The values were learned with the parameter settings $\alpha = 0.05$ and $\lambda = 1$. The curves are for the predictions about the red, blue, and green observation bits. The agent learns early in training, but none of the predictions can be made perfectly at all times.

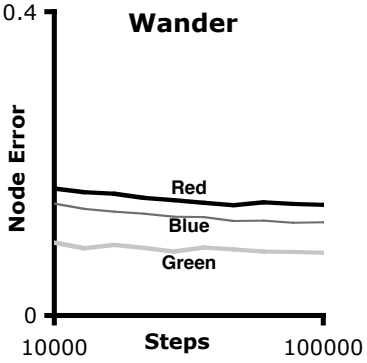


Figure 4.7: Individual node errors for the **Wander** node averaged over 30 runs of 100,000 steps. The values were learned with the parameter settings $\alpha = 0.05$ and $\lambda = 1$. The curves are for the predictions about the red, blue, and green observation bits. There is a slight improvement in prediction, but the outcome of the **Wander** option cannot be learned perfectly.

due to the state abstraction performed by the OTD network. When facing each direction, states are abstracted into groups of the states that face North, South, East, or West. Because the agent cannot distinguish its exact position, the predictions for **Wander** will differ from the oracle values. Also, the agent's predictions can be close to the oracle value, but any difference, however minimal, will contribute error to the system. The prediction error for the red, blue, and green observation bits is shown in Figure 4.7. Over time, there is a gradual, but minimal improvement.

4.5.6 Maintaining Direction

One goal of this research is to connect sensations to high-level concepts. An example of a concept learned from data in the grid-world experiments is that of *direction*—a concept

that clearly emerges (the agent uses its own predictions to keep track of the direction that it is facing).

The concept of direction is demonstrated in Figure 4.8. After 250,000 steps of training, the agent was manually maneuvered into the position shown at time $t = 1$. The agent was then spun clockwise (R) for six full rotations from time $t = 1$ to $t = 25$. The predictions were recorded for the nodes corresponding to **Leap** and **Leap**-L-**Leap**. These predictions appear as bar diagrams in the figure. As the agent rotates the correct predictions are maintained even though the only information received from the environment is the activation of the white observation bit. In fact, the agent could continue to spin clockwise (or counterclockwise) indefinitely and the predictions would remain correct because the network's predictions from the current step determine the predictions on the next time step.

Of particular interest is the prediction for **Leap** at $t = 4$, which is non-zero for both blue and green. As discussed in the Section 4.5.5, the agent cannot know exactly which row it is in. Rather, the agent knows that with some probability executing the **Leap** option until termination will result in an observation of blue and with a lesser probability, the option's execution until termination will result in an observation of green. The actual prediction values are close to $\frac{5}{6}$ for blue and $\frac{1}{6}$ for green. This ratio corresponds to the six possible rows in which the agent could be located.

The predictions in the rightmost column (predictions about the sequence **Leap**-L-**Leap**) are correct in all cases. There is no need to make probabilistic predictions about the green and blue observations because the sequence always moves the agent into one of the corners. There is therefore no ambiguity as to the agent's row or column.

For time steps $t = 26, \ldots, 29$ the agent is manually maneuvered to the top of the environment by forcing it to take three steps forward and rotate left. At this point, because the agent observed orange at $t = 28$ (identifying that it is in the top row), it can make correct predictions about the green bit on the subsequent step. At $t = 29$ the agent correctly predicts that if the **Leap** option were to be executed, green would be observed (and blue would not be observed).

4.6 Discussion and Conclusions

In this chapter we have investigated the first on-policy algorithm for learning OTD networks. A forward-view algorithm was re-expressed as an incremental algorithm; the incremental

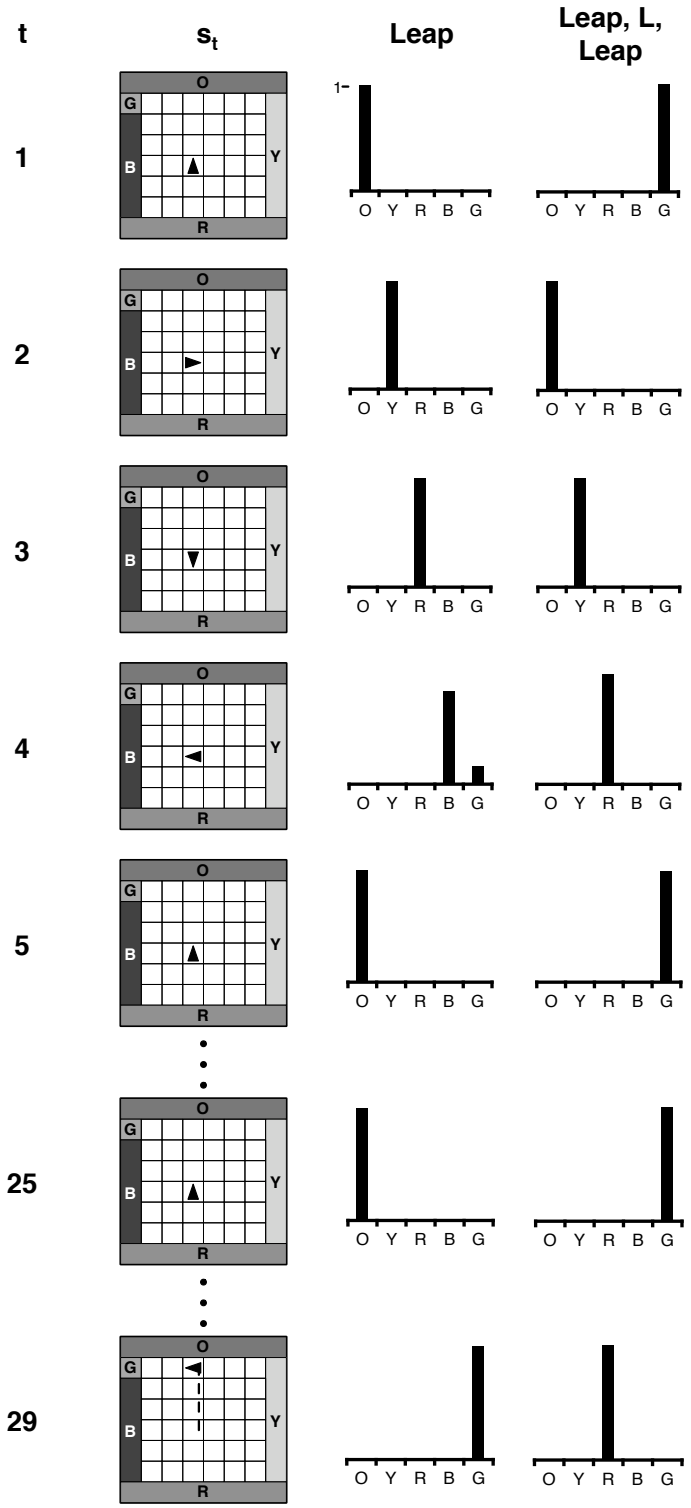


Figure 4.8: A sample 29-step trajectory in the grid world. From $t = 1$ until $t = 25$ the agent is rotated clockwise. From $t = 26$ to $t = 29$ the agent takes 3 step-forward actions and one rotate-left action. The first column is the relative time step (after 250,000 steps of training). The second column is an illustration of the agent's location in the world. The third and fourth columns are the node predictions for the **Leap** option and the sequence of options **Leap**-L-**Leap**. The bar chart indicates the magnitude of the prediction for orange (O), yellow (Y), red (R), blue (B), and green (G).

algorithm was used to learn predictions for an OTD network in a grid-world. While certain predictions, which do not depend on knowing the agent's exact position in the grid world, can be learned perfectly, others cannot, but in all cases prediction error decreases over time. Finally, an agent is shown to learn the concept of *direction* in the grid world.

In the presented experiments, temporal abstraction allows concepts, such as the concept of direction, to be learned. But is it not true that the example grid world could be modeled as a series of single time-step transitions? It is indeed possible to make an accurate model by chaining together multiple step-forward predictions instead of using the **Leap** option. However, by using the **Leap** option the OTD network is not constrained to any particular environment size. Given a world that has the same color structure as the grid world of Figure 4.2, an OTD network with the exact same structure as Figure 4.3 can be used to model the world, regardless of the world's size. A TD network, on the other hand, would need additional predictions to model the growing world. Limited experiments show that by incrementally expanding the size of the grid world[4], an agent can make correct long-term prediction in worlds as large as 100x100 with the same OTD network used in the experiments in Section 4.5.

Questions also surround the robustness of the OTD network learning algorithm in the presence of stochasticity. Experiments were performed with a probability of "slipping" when the forward action is selected (with some probability the step-forward action had no effect). The **Leap** option continued to make the correct predictions in this case, though training times increased as the slipping probability increased. The agent was able to make correct predictions because the agent was still executing the **Leap** option to termination regardless of the slip. The temporally-abstract nature of the **Leap** option leads the option to cope with forward-slippage.

A slipping probability was then incorporated in the rotate actions. In this case, the concept of direction (as in Section 4.5.6) was still present, but the possibility of slipping was incorporated in the predictions. As the agent was continuously rotated, predictions became less and less certain for the **Leap** node since there was a probability that the agent slipped during the rotation. Eventually, after enough rotations, the agent's predictions became inaccurate. From then on, correct predictions could not be made until the agent ran into a colored wall and was thus able to re-orient itself. What the agent learns in the presence

[4]Incremental expansion is used to accelerate learning. Because the growing worlds maintain a similar structure, the predictions learned in an OTD network can be used as initial values for training in a larger world.

of slippage is akin to how a person would deal with being blindfolded and spun in circles. At some point, the person would lose track of the direction that they are facing. Upon removing the blindfold, the person would be able to regain their bearings.

Chapter 5

Universal Off-policy Learning

This chapter explores the issue of learning multiple option outcomes from a single stream of experience. In the previous chapter predictions were learned by executing an option's policy from initiation until termination. The downside to this *on-policy* learning strategy is that the agent can only learn about one option at a time—the one whose policy is being followed. A more efficient use of data is to simultaneously learn about all policies that are in any way similar to the agent's behavior. Learning about a policy other than the one being followed is known as *off-policy* learning. However, off-policy learning introduces potential instabilities when combined with function approximation and temporal-difference methods (Baird, 1995). Precup, Sutton and Dasgupta presented the first provably sound algorithm for off-policy temporal-difference learning with linear function approximation (2001). In their algorithm, potential instabilities were counteracted by using importance-sampling corrections to condition the weight updates. The work of Precup et al. is extended in this thesis to the off-policy learning of option models. In order to directly study off-policy learning, a TD network's question network is used to specify the predictions, but the predictions are not used as state. Instead, the agent observes a feature vector which is emitted by the environmental state.

5.1 Off-policy Learning

The outcome of a single option can be learned by repeatedly following the option's policy until termination, but how should the outcomes of multiple options be learned? One possibility is to choose an option to learn about and follow the corresponding policy until termination; a better alternative is to choose a behavior policy and learn about all options with similar policies. As the number of options increases, or as the time until termination

increases, the former, on-policy learning, becomes less practical because the amount of data becomes small in proportion to the number of outcomes that the agent is trying to predict. It is generally more efficient to learn about multiple ways of behaving from a single stream of data.

Consider an agent trained in an on-policy manner: options are chosen, then followed until termination. With an on-policy algorithm the agent learns only about the option it is following; with an off-policy learning algorithm, the agent follows one option's policy, learns about that option, but also learns about every option with a similar policy. In this training scheme, off-policy learning evidently allows the agent to use data more efficiently.

Off-policy learning is an issue of interest in the reinforcement-learning community. For example, Q-Learning is an off-policy algorithm (Watkins, 1989) in which the agent learns about the optimal policy while following an ϵ-greedy policy (the agent chooses a random action with probability ϵ and chooses the optimal action otherwise). While there have been many successes with Q-learning, examples exist demonstrating that it can diverge when combined with function approximation (Baird, 1995). This instability is a general issue when off-policy learning is combined with function approximation and TD methods. Precup, Sutton and Dasgupta introduced the first provably sound off-policy algorithm for temporal-difference learning with linear function approximation (2001). The algorithm incorporated importance-sampling corrections to condition weight updates. Their new off-policy TD(λ) algorithm was shown to have the same expected updates as the on-policy TD(λ) algorithm—an algorithm that was guaranteed to converge when using linear function approximation (Bertsekas & Tsitsiklis, 1996). In this thesis, the Precup et al's off-policy algorithm is used as a basis for a new off-policy algorithm for the learning of option models. The new algorithm provably to makes the same expected updates as the on-policy algorithm for learning option models.

It is important to note that in this chapter a temporal-difference network is used to specify predictions, but the predictions are not used as a state representation. This important distinction is made in order to study off-policy learning separately from OTD-network learning. A possible complication with using the predictions of a TD network as a state representation is that these predictions are learned. For a given environmental state, the agent may receive a completely different set of features depending on the amount of training conducted by the agent. Theoretical guarantees have not been made for the case where predictions are used as state. Off-policy learning with non-stationary features (learned

predictions) is revisited in Chapter 6.

An agent predicts the outcome of following a *target* policy, $\pi(\cdot, \cdot)$ until termination and actions are chosen according to a *behavior* policy, $b(\cdot, \cdot)$. In on-policy learning, $\pi(s, a) = b(s, a)$, $\forall\, s, a$; in off-policy learning, $\exists s, a$ such that $\pi(s, a) \neq b(s, a)$. An importance sampling factor $\rho(s, a) = \frac{\pi(s,a)}{b(s,a)}$ corrects for the difference in the frequency of action selection between the target policy and the behavior policy.

Intuitively, the importance-sampling factor leads to large weight updates when the agent chooses an action that is commonly chosen by the target policy but rarely chosen by the behavior policy. Conversely, an action that is rarely selected by the target policy but frequently selected by the behavior policy results in smaller weight updates. Importance-sampling corrections have been used to successfully address the issue of off-policy learning in several papers (Precup, Sutton, & Singh, 2000; Precup, Sutton, & Dasgupta, 2001; Precup, Sutton, Paduraru, Koop, & Singh, 2005).

5.2 Algorithm Derivation

This section presents the derivation of an incremental update rule for the off-policy learning of option models—similar to the derivation found in Section 4.4. A forward view for the off-policy learning of option models is defined, then a backward view with the same expected updates is derived. As in Chapter 4, the agent attempts to learn the expected value of the outcome of an option (Equation 4.11). Unlike Chapter 4, the off-policy algorithm presented in this chapter has the following characteristics:

- An importance sampling correction ρ_t accounts for differences between the behavior policy and the target policy;
- The condition c_t is removed from the weight update equation;
- κ_t accumulates importance-sampling corrections and accounts for the possibility that the option could be initiated at multiple states over the course of its single execution;
- The feature vector $\boldsymbol{\phi}$ is emitted by the environment rather than being constructed by the agent.

5.2.1 The Forward View

As in Section 4.4.1 an oracle provides future targets $z_{t+1}, z_{t+2}, \ldots, z_{t+n}$. These targets define the n-step outcomes:

$$\bar{Z}_t^{(n)} = \rho_t(\beta_{t+1} z_{t+1} + (1 - \beta_{t+1})\bar{Z}_{t+1}^{(n-1)}) \tag{5.1}$$

where ρ_t is the importance-sampling correction at time t and the base case is $\bar{Z}_t^{(0)} = y_t$.

Equation 5.1 is similar to the forward view defined in Equation 4.12—the key difference being the inclusion of the importance sampling correction ρ_t. The extensive form of each n-step outcome is:

$$\begin{aligned}
\bar{Z}_t^{(1)} &= \rho_t(\beta_{t+1} z_{t+1} + (1 - \beta_{t+1}) y_{t+1}) \\
\bar{Z}_t^{(2)} &= \rho_t\big(\beta_{t+1} z_{t+1} + (1 - \beta_{t+1})[\rho_{t+1}(\beta_{t+2} z_{t+2} + (1 - \beta_{t+2}) y_{t+2})]\big) \\
\bar{Z}_t^{(3)} &= \rho_t\Big(\beta_{t+1} z_{t+1} + (1 - \beta_{t+1})\big[\rho_{t+1}\big(\beta_{t+2} z_{t+2} + (1 - \beta_{t+2}) \\
&\qquad\qquad [\rho_{t+2}(\beta_{t+3} z_{t+3} + (1 - \beta_{t+3}) y_{t+3})]\big)\big]\Big) \\
&\vdots
\end{aligned}$$

As in Section 4.4, the n-step outcomes are blended to from the lambda outcome:

$$\bar{Z}_t^{\lambda} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} \bar{Z}_t^{(n)}, \tag{5.2}$$

and the weight updates made over the course of an option's execution is:

$$\Delta\bar{\theta} = \sum_{t=0}^{T} \alpha(\bar{Z}_t^{\lambda} - y_t) \nabla_\theta y_t \kappa_t, \tag{5.3}$$

where the quantity κ_t keeps track of the product of importance-sampling corrections over the course of an option's execution. κ_t is necessary because the agent must correct for the ratio between the entire sequence of actions being taken under the target policy and the sequence of actions being taken under the behavior policy. κ_t is defined as:

$$\kappa_t = \sum_{i=0}^{t} g_i \prod_{j=i}^{t-1} \rho_j \prod_{j=i+1}^{t} (1 - \beta_j), \tag{5.4}$$

in which the value g_i incorporates *restarts* into the equation.

5.2.2 Restarting an Option During Execution

Over the course of an option's execution, the agent may pass through multiple states that belong to the option's initiation set, $\mathcal{I}$. It is thus possible that an option could be initiated

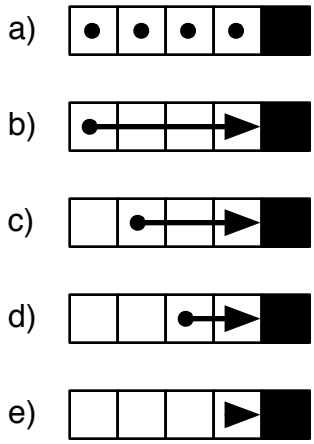


Figure 5.1: While facing east, an agent attempting to predict the outcome of stepping forward until reaching a wall can initiate the option from any of the dotted states in (a). If the agent starts in the leftmost state, then the trajectory followed in (b) passes through states in which the option could be initiated. The quantity g_i in Equation 5.4 can account for the initiation in each state in the option's initiation set.

from any of these states. Figure 5.1 demonstrates a situation in which an agent may pass through multiple states from an option's initiation set. Suppose the agent is facing East and is learning a prediction for stepping forward until reaching the wall. In Figure 5.1a, the dots identify the states in which the option can be initiated. If the agent begins in the leftmost state, then Figure 5.1b shows a trajectory that follows the option policy until termination. Over the course of this trajectory, the option *could* be initiated from each visited state , and from each possible initial state, the option would be followed until termination (Figures 5.1c-5.1e).

The quantity g_i in Equation 5.4 allows restarts to be included in the forward-view equations. A possible setting for g_i is to let $g_0 = 1$ and $g_t = 0, \forall t \geq 1$. This is the case when an option is initiated only at the beginning of its execution. An agent that follows the trajectory in Figure 5.1b assigns credit to each state in the trajectory.

Another possible setting is to let $g_i = 1$ for all states in the option's initiation set. The weight updates would then account for the possibility of starting from each of state in the initiation set. In the example, an agent that follows the trajectory in Figure 5.1b assigns credits for states along the trajectory, but also assigns additional credit to the states in the trajectories shown in Figure 5.1c, Figure 5.1d, and Figure 5.1e. Thus, the state adjacent to the wall receives credit for four visits while the leftmost state receives credit for a single visit. In the experiments in both this chapter and Chapter 6, $g_i = 1$ whenever the option can be initiated.

Dealing with restarts during the execution of an option was originally introduced in Precup, Sutton, & Dasgupta (2001) where it is shown that for some distribution of starting states, the algorithm with restarts will have the same updates as the algorithm without restarts.

5.2.3 Forward and Backward View Equivalence

The backwards view of the off-policy algorithm for learning option models begins with the re-expression of the error term from Equation 5.3:

$$
\begin{aligned}
\bar{Z}_t^\lambda - y_t &= -y_t + (1-\lambda)\lambda^0 \bar{Z}_t^{(1)} + (1-\lambda)\lambda^1 \bar{Z}_t^{(2)} + (1-\lambda)\lambda^2 \bar{Z}_t^{(3)} + \dots \\
&= -y_t \\
&\quad + (1-\lambda)\lambda^0 [\rho_t(\beta_{t+1} z_{t+1} + (1-\beta_{t+1}) y_{t+1})] \\
&\quad + (1-\lambda)\lambda^1 \big[\rho_t\big(\beta_{t+1} z_{t+1} + (1-\beta_{t+1})[\rho_{t+1}(\beta_{t+2} z_{t+2} + (1-\beta_{t+2}) y_{t+2})]\big)\big] \\
&\quad + (1-\lambda)\lambda^2 \Big[\rho_t\Big(\beta_{t+1} z_{t+1} + (1-\beta_{t+1})\big[\rho_{t+1}\big(\beta_{t+2} z_{t+2} + (1-\beta_{t+2}) \\
&\qquad\qquad [\rho_{t+2}(\beta_{t+3} z_{t+3} + (1-\beta_{t+3}) y_{t+3})]\big)\big]\Big)\Big] \\
&\quad + \cdots \\
\\
&= -y_t \\
&\quad + \lambda^0 [\rho_t(\beta_{t+1} z_{t+1} + (1-\beta_{t+1}) y_{t+1}) - \lambda \rho_t(\beta_{t+1} z_{t+1} + (1-\beta_{t+1}) y_{t+1})] \\
&\quad + \lambda^1 \big[\rho_t\big(\beta_{t+1} z_{t+1} + (1-\beta_{t+1})[\rho_{t+1}(\beta_{t+2} z_{t+2} + (1-\beta_{t+2}) y_{t+2})]\big) \\
&\qquad\qquad - \lambda \rho_t\big(\beta_{t+1} z_{t+1} + (1-\beta_{t+1})[\rho_{t+1}(\beta_{t+2} z_{t+2} + (1-\beta_{t+2}) y_{t+2})]\big)\big] \\
&\quad + \lambda^2 \Big[\rho_t\Big(\beta_{t+1} z_{t+1} + (1-\beta_{t+1})\big[\rho_{t+1}\big(\beta_{t+2} z_{t+2} + (1-\beta_{t+2}) \\
&\qquad\qquad\qquad [\rho_{t+2}(\beta_{t+3} z_{t+3} + (1-\beta_{t+3}) y_{t+3})]\big)\big]\Big) \\
&\qquad\qquad - \lambda \rho_t\Big(\beta_{t+1} z_{t+1} + (1-\beta_{t+1})\big[\rho_{t+1}\big(\beta_{t+2} z_{t+2} + (1-\beta_{t+2}) \\
&\qquad\qquad\qquad [\rho_{t+2}(\beta_{t+3} z_{t+3} + (1-\beta_{t+3}) y_{t+3})]\big)\big]\Big)\Big] \\
&\quad + \cdots
\end{aligned}
$$

$$
\begin{aligned}
= \quad & \lambda^0[\rho_t(\beta_{t+1}z_{t+1} + (1-\beta_{t+1})y_{t+1} - y_t] \\
& + \lambda^1\big[\rho_t\big(\beta_{t+1}z_{t+1} + (1-\beta_{t+1})[\rho_{t+1}(\beta_{t+2}z_{t+2} + (1-\beta_{t+2})y_{t+2})]\big) \\
& \qquad - \rho_t\big(\beta_{t+1}z_{t+1} + (1-\beta_{t+1})y_{t+1}\big)\big] \\
& + \lambda^2\Big[\rho_t\Big(\beta_{t+1}z_{t+1} + (1-\beta_{t+1})\big[\rho_{t+1}\big(\beta_{t+2}z_{t+2} + (1-\beta_{t+2})[\rho_{t+2} \\
& \qquad (\beta_{t+3}z_{t+3} + (1-\beta_{t+3})y_{t+3})]\big)\big]\Big) \\
& \qquad - \rho_t\Big(\beta_{t+1}z_{t+1} + (1-\beta_{t+1})\big[\rho_{t+1}\big(\beta_{t+2}z_{t+2} + (1-\beta_{t+2})y_{t+2}\big)\big]\Big)\Big] \\
& + \cdots \\
\\
= \quad & \lambda^0(\bar{Z}_t^{(1)} - y_t) \\
& + \lambda^1(\bar{Z}_{t+1}^{(1)} - y_{t+1})\rho_t(1-\beta_{t+1}) \\
& + \lambda^2(\bar{Z}_{t+2}^{(1)} - y_{t+2})\rho_t\rho_{t+1}(1-\beta_{t+1})(1-\beta_{t+2}) \\
& + \cdots \\
& = \sum_{i=t}^{\infty}\lambda^{i-t}\delta_i\prod_{j=t}^{i-1}\rho_j\prod_{j=t+1}^{i}(1-\beta_j)
\end{aligned}
$$

where

$$
\begin{aligned}
\delta_i &= (\bar{Z}_i^{(1)} - y_i) \\
&= \rho_i(\beta_{i+1}z_{i+1} + (1-\beta_{i+1})y_{i+1}) - y_i
\end{aligned}
$$

The new definition of the error term $\bar{Z}_t^{\lambda} - y_t$ is substituted into the sum of weight updates (Equation 5.3):

$$
\begin{aligned}
\sum_{t=0}^{T}\alpha(\bar{Z}_t^{\lambda} - y_t)\nabla_\theta y_t\kappa_t &= \sum_{t=0}^{T}\alpha\nabla_\theta y_t\kappa_t\sum_{i=t}^{T}\lambda^{i-t}\delta_i\prod_{j=t}^{i-1}\rho_j\prod_{j=t+1}^{i}(1-\beta_j) \\
&= \sum_{t=0}^{T}\alpha\delta_t\sum_{i=0}^{t}\lambda^{t-i}\nabla_\theta y_i\kappa_i\prod_{j=i}^{t-1}\rho_j\prod_{j=i+1}^{t}(1-\beta_j) \\
&= \sum_{t=0}^{T}\alpha\delta_t e_t
\end{aligned}
$$

$$
\text{where } e_t = \sum_{i=0}^{t}\lambda^{t-i}\nabla_\theta y_i\kappa_i\prod_{j=i}^{t-1}\rho_j\prod_{j=i+1}^{t}(1-\beta_j) \tag{5.5}
$$

It can be demonstrated that the recursive definitions of κ_t and e_t (shown next) are equivalent to Equation 5.4 and Equation 5.5, respectively. The recursive definitions of κ_t

and e_t are:

$$\kappa_0 = g_0$$

$$\kappa_t = \rho_{t-1}\kappa_{t-1}(1-\beta_t) + g_t \tag{5.6}$$

$$e_0 = \nabla_\theta y_0 \kappa_0$$

$$e_t = \lambda(1-\beta_t)\rho_{t-1}e_{t-1} + \kappa_t \nabla_\theta y_t \tag{5.7}$$

where the value of g_0 is generally 1 (see Section 5.2.2 for an explanation). The recursive definitions are shown to be equivalent to the forward equations via induction.

Theorem 5.2.1

$$\kappa_t = \sum_{i=0}^{t} g_i \prod_{j=i}^{t-1} \rho_j \prod_{j=i+1}^{t} (1-\beta_j) = \rho_{t-1}\kappa_{t-1}(1-\beta_t) + g_t \tag{5.8}$$

$$k_0 = g_0$$

Proof The bases case are equivalent by definition:

$$\kappa_0 = \sum_{i=0}^{0} g_i \prod_{j=i}^{-1} \rho_j \prod_{j=i+1}^{0} (1-\beta_j) = g_0.$$

(If the initial index of a product is larger than the upper bound then the term is omitted from the equation.)

Next, assuming that Equation 5.8 is true for κ_t:

$$\begin{aligned}
\kappa_{t+1} &= \rho_t\kappa_t(1-\beta_{t+1}) + g_{t+1} \\
&= \rho_t\Big(\sum_{i=0}^{t} g_i \prod_{j=i}^{t-1} \rho_j \prod_{j=i+1}^{t} (1-\beta_j)\Big)(1-\beta_{t+1}) + g_{t+1} && \text{Equation 5.8} \\
&= \Big(\sum_{i=0}^{t} g_i \prod_{j=i}^{t} \rho_j \prod_{j=i+1}^{t+1} (1-\beta_j)\Big) + g_{t+1} && c\sum_{i}^{n} a = \sum_{i}^{n} ca \\
&= \sum_{i=0}^{t+1} g_i \prod_{j=i}^{t} \rho_j \prod_{j=i+1}^{t+1} (1-\beta_j) && g_{t+1}\prod_{j=t+1}^{t} \rho_j \prod_{j=(t+1)+1}^{t+1} (1-\beta_j) = g_{t+1}
\end{aligned}$$

∎

Theorem 5.2.2

$$e_t = \sum_{i=0}^{t} \lambda^{t-i}\nabla_\theta y_i \kappa_i \prod_{j=i}^{t-1} \rho_j \prod_{j=i+1}^{t} (1-\beta_j) = \lambda(1-\beta_t)\rho_{t-1}e_{t-1} + \kappa_t\nabla_\theta y_t \tag{5.9}$$

$$e_0 = \nabla_\theta y_0 \kappa_0$$

Proof The bases case are equivalent by definition:

$$e_0 = \sum_{i=0}^{0} \lambda^{0-i} \nabla_\theta y_i \kappa_i \prod_{j=i}^{0-1} \rho_j \prod_{j=i+1}^{0} (1 - \beta_j) = \nabla_\theta y_0 \kappa_0.$$

Next, assuming that Equation 5.9 is true for e_t:

$$\begin{aligned}
e_{t+1} &= \lambda(1-\beta_{t+1})\rho_t e_t + \kappa_{t+1}\nabla_\theta y_{t+1} \\
&= \lambda(1-\beta_{t+1})\rho_t \Bigg(\sum_{i=0}^{t} \lambda^{t-i} \nabla_\theta y_i \kappa_i \prod_{j=i}^{t-1} \rho_j \prod_{j=i+1}^{t} (1-\beta_j) \Bigg) + \kappa_{t+1}\nabla_\theta y_{t+1} && \text{Equation 5.9} \\
&= \Bigg(\sum_{i=0}^{t} \lambda^{(t+1)-i} \nabla_\theta y_i \kappa_i \prod_{j=i}^{t} \rho_j \prod_{j=i+1}^{t+1} (1-\beta_j) \Bigg) + \kappa_{t+1}\nabla_\theta y_{t+1} && c\sum_{i}^{n} a = \sum_{i}^{n} ca \\
&= \sum_{i=0}^{t+1} \lambda^{(t+1)-i} \nabla_\theta y_i \kappa_i \prod_{j=i}^{t} \rho_j \prod_{j=i+1}^{t+1} (1-\beta_j) && \lambda^{(t+1)-(t+1)} \nabla_\theta y_{t+1} \kappa_{t+1} \prod_{j=t+1}^{t} \rho_j \\
& && \prod_{j=(t+1)+1}^{t+1} (1-\beta_j) = \kappa_{t+1}\nabla_\theta y_{t+1}
\end{aligned}$$

∎

As in Chapter 4, we do not have y_{t+1} when computing δ_t. Instead, $\tilde{y}_{t+1}$ (Equation 4.20) is used as an approximation of the prediction on the next time step. The temporal-difference error, δ_t is therefore calculated as

$$\delta_t \quad = \quad \rho_t(\beta_{t+1} z_{t+1} + (1-\beta_{t+1})\tilde{y}_{t+1}) - y_t. \tag{5.10}$$

The last difference between the on-policy algorithm presented in Chapter 4 and the off-policy algorithm of this chapter is the manner in which the weight vector $\boldsymbol{\theta}_t$ is updated. The condition variable c_t is not used in the off-policy learning algorithm; thus the weight update can be described on an element-by-element basis as:

$$\theta^{ij}_{t+1} = \theta^{ij}_t + \alpha \delta^i_t e^{ij}_t. \tag{5.11}$$

The order of computation is as follows:

$$y_t \ \kappa_t \ \mathbf{e}_t \ a_t \ \rho_t \ \boldsymbol{\phi}_{t+1} \ \tilde{y}_{t+1} \ \beta_{t+1} \ z_{t+1} \ \delta_t \ \boldsymbol{\theta}_{t+1} \ y_{t+1} \tag{5.12}$$

Pseudocode for implementing this algorithm can be found in Algorithm 2.

5.2.4 Convergence

A proof developed by Precup, Sutton, Paduraru, Koop & Singh (2005) is adapted to show that the on-policy algorithm (Chapter 4) and off-policy algorithm (Chapter 5) share the same expected updates.

Algorithm 2 The universal off-policy learning algorithm.

Initialize $\mathbf{y}_0, \mathbf{E}_0, \boldsymbol{\theta}_0, \boldsymbol{\beta}_0, \boldsymbol{\kappa}_0, \boldsymbol{\rho}_0$

for $t = 1, 2, \ldots$

  Take action a_t; receive feature vector $\boldsymbol{\phi}_{t+1}$

  Update product of importance sampling corrections: $\boldsymbol{\kappa}_t = \boldsymbol{\rho}_{t-1}\boldsymbol{\kappa}_{t-1}(1 - \beta_t) + \mathbf{g}_t$

  Compute importance sampling corrections: $\boldsymbol{\rho}_t = \boldsymbol{\rho}(a_t)$

  Update trace matrix: $\mathbf{E}_t = \lambda(1 - \boldsymbol{\beta}_t)\boldsymbol{\rho}_{t-1}\mathbf{E}_{t-1} + \boldsymbol{\kappa}_t \nabla_{\boldsymbol{\theta}_t} \mathbf{y}_t$

  Compute interim predictions: $\tilde{\mathbf{y}}_{t+1} = \mathbf{u}(\boldsymbol{\phi}_{t+1}, \boldsymbol{\theta}_t)$

  Check for termination: $\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}(o_{t+1}, \mathbf{y}_t)$

  Update target values: $\mathbf{z}_t = \mathbf{z}(o_{t+1}, \tilde{\mathbf{y}}_{t+1})$

  Compute error: $\boldsymbol{\delta}_t = \boldsymbol{\rho}(\boldsymbol{\beta}_{t+1}\mathbf{z}_{t+1} + (1 - \boldsymbol{\beta}_{t+1})\tilde{\mathbf{y}}_{t+1}) - \mathbf{y}_t$

    (multiplications are component-wise)

  Update weights: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\boldsymbol{\delta}_t\mathbf{E}_t$

  Update predictions: $\mathbf{y}_{t+1} = \mathbf{u}(\boldsymbol{\phi}_{t+1}, \boldsymbol{\theta}_{t+1})$

end for

Before the off-policy algorithm is discussed, the on-policy algorithm must be shown to converge. In the on-policy algorithm introduced in Chapter 4, predictions were used as a features for the representation; in this chapter, the algorithm is modified to use the stationary feature-vector which is generated as a function of the environmental state ($\boldsymbol{\phi}_t = \boldsymbol{\phi}(s_t)$). The convergence result of Bertsekas and Tsitsiklis (1996, p. 309) for episodic TD(λ) with linear function approximation can be directly applied to the modified on-policy algorithm. The option model being learned is a special case of episodic TD(λ) where:

- the option's initiation at $t = 0$ corresponds to the initiation of an episode;
- the option terminates at $t = T$ ($\beta(s_T) = 1.0$), corresponding to the termination of an episode;
- the reward $r_t = 0, \forall t < T$ and $r_T = z_T$.

Next, to apply Precup et al.'s proof it must first be shown that the expected values of the n-step outcomes are equivalent under both the target policy and the behavior policy.

Theorem 5.2.3 *For any initial state s,*

$$E_b[\bar{Z}_t^{(n)}|s] = E_\pi[Z_t^{(n)}|s], \forall n. \tag{5.13}$$

Proof The bases case holds by definition:

$$\bar{Z}_t^{(0)} = Z_t^{(0)} = y_t.$$

Next, assuming that Equation 5.13 is true for $n-1$:

$$\begin{aligned}
E_b[\bar{Z}_t^{(n)}|s] &= \sum_a b(s,a) \sum_{s'} \mathcal{P}_{ss'}^a \rho(s,a) \Big[\beta(s') z_{t+1} + (1-\beta(s')) E_b[\bar{Z}_{t+1}^{(n-1)}|s]\Big] && \text{Equation 5.1} \\
&= \sum_a \sum_{s'} \mathcal{P}_{ss'}^a b(s,a) \frac{\pi(s,a)}{b(s,a)} \Big[\beta(s') z_{t+1} + (1-\beta(s')) E_b[\bar{Z}_{t+1}^{(n-1)}|s]\Big] && \text{Definition of } \rho(s,a) \\
&= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}_{ss'}^a \Big[\beta(s') z_{t+1} + (1-\beta(s')) E_b[\bar{Z}_{t+1}^{(n-1)}|s]\Big] && c\sum_i^n a = \sum_i^n ca \\
&= E_\pi[Z_t^{(n)}|s] && \text{Equation 5.1}
\end{aligned}$$

∎

Equation 5.13 implies that $E_b[\bar{Z}_t^\lambda|s] = E_\pi[Z_t^\lambda|s]$. Having established the equivalence of the λ-returns, the proof by Precup, et al. can be directly applied to show that the expected values of the weight updates are identical between the on-policy and off-policy algorithms:

$$E_b[\Delta\bar{\theta}_t|s_0] = E_\pi[\Delta\theta_t|s_0], \tag{5.14}$$

where the option is initiated in the same state s_0 for both the on-policy and off-policy algorithms.

In a limited set of experiments, an on-policy learning agent was shown empirically to have the same expected weight updates as an off-policy learning agent. For the on-policy agent, weight changes were accumulated over the course of a **Leap** option's execution according to Equation 4.15. The total update was equivalent to the expected weight update because the environment was deterministic. The off-policy agent learned the expected value of the total weight update (Equation 5.3) for the **Leap** option by following 50,000 different trajectories generated from the behavior policy $b = \{p(\cdot, F) = 0.5, p(\cdot, L) = 0.25, p(\cdot, R) = 0.25\}$. Each trajectory lasted until the agent either terminated (reached the wall) or diverged (took an action other than F). The off-policy agent did not learn about restarting during a trajectory ($g_0 = 1, g_t = 0, \forall t \geq 1$).

At the initiation of an option both agents were placed in the same environmental state in the grid world (Figure 4.2), λ was fixed at 1.0, and all weights (and thus predictions) were initialized to 0. The experiment was repeated for multiple starting state and both agents had the same expected weight updates for each starting state. These experiments

suggest that Equation 5.14 holds, but experimentation in stochastic environments would be of interest.

However, while the on-policy algorithm and the off-policy algorithm have the same expected sum of updates, a problem exists with the variance of the updates. A condition for the convergence of the on-policy algorithm (and thus for the off-policy algorithm as well) is that the variance must be bounded. When learning about option outcomes, if an option can be guaranteed to terminate in a finite amount time, then the variance will be bounded because the weight update will be computed from a finite number of bounded quantities.

However, the product of importance-sampling corrections is accumulated in κ, which can become large over the course of an option's execution. For instance, suppose an action, a, is twice as likely to be taken under the target policy as compared to the behavior policy. The product of importance-sampling corrections doubles every time a is selected by the behavior policy. Because the corrections accumulate over time, the total importance-sampling correction grows exponentially in the number of times that a is selected over the course of a single option's execution. In the experiments presented in Section 5.3, a small step-size parameter, α, is used to counteract the large variance. This is not an entirely satisfactory solution to the problem of large variance, and other possible solutions are discussed in Section 7.1.2.

5.3 Tiled Gridworld Experiments

This section presents results from experiments conducted in the grid world of Figure 4.2 (the grid world can also be seen in Figure 5.2). There is one difference between the grid world in this section and the one from the previous chapter, and that is the agent's observation vector. In this section, the environment emits a 41-element binary feature vector. As before, the first bit is a bias term, and the next six bits correspond to the six possible colors that the agent can observe. The next four elements, however, correspond to compass directions, where the bit corresponding to the agent's current direction will have a value of 1 and the other three will be 0. The final 30 elements indicate the agent's position among a set of horizontal and vertical tilings that have been overlaid on the environment. Figure 5.2 illustrates the horizontal tilings: two of width 2 and three of width 3. There is an element in the feature vector corresponding to each of the 15 horizontal tiles. If the agent is in a tile, then the corresponding bit has a value of 1, otherwise the bit is 0. Vertical tilings are constructed similarly to the horizontal tiles, but rotated 90°.

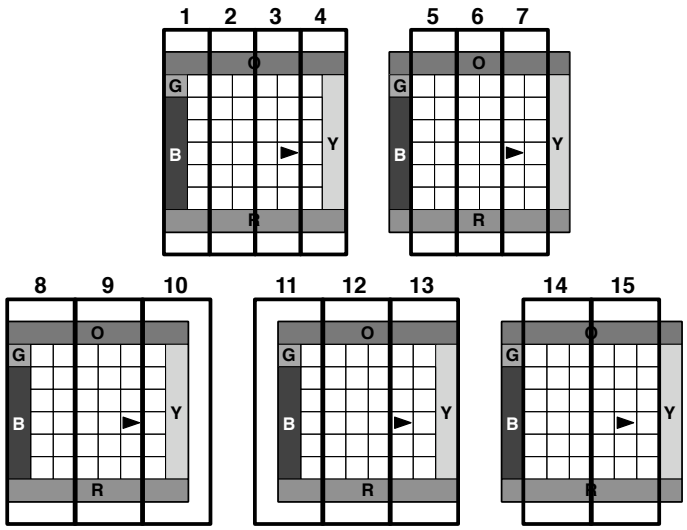


Figure 5.2: Features are obtained by partitioning the the grid world from Figure 4.2 in five different tilings. Two tilings are of width 2 and three tilings are of width 3. For each of the five tilings, the agent will be located in one tile and the feature corresponding to that tile will have a value of 1 while the other feature(s) will be 0. For the agent position pictured in this figure, the following tiles will be active: 3, 7, 9, 13, and 15. A similar set of five vertical tilings exists as well.

The feature vector corresponding to the agent's position in Figure 5.2 is thus

$$\Big\{\underbrace{1}_{bias}\ \underbrace{1\,0\,0\,0\,0\,0}_{color}\ \underbrace{0\,1\,0\,0}_{direction}\ \underbrace{0\,0\,1\,0\,0\,0\,1\,0\,1\,0\,0\,0\,1\,0\,1}_{horizontal\ tilings}\ \underbrace{0\,0\,1\,0\,0\,1\,0\,0\,1\,0\,0\,1\,0\,0\,1}_{vertical\ tilings}\Big\}$$

where the active bit in the color selection denotes an observation of white, the active bit in the direction section indicates that the agent is facing East, and the active bits in the tilings correspond to the tiles in which the agent is located.

A question network identical to the one pictured in Figure 4.3 defines the agent's predictions. There are five connected components, each identical in structure, but asking questions about a different color. Each connected component consists of the prediction of an observation bit after one of the following nine action sequences: F, L, R, **Leap**, L-**Leap**, R-**Leap**, **Leap**-L-**Leap**, **Leap**-R-**Leap**, and **Wander**.

5.3.1 Parameter Study

The first set of experiments were designed to determine the best parameter settings for α and λ. In these experiments, the agent used the off-policy learning algorithm presented in Section 5.2.3 to learn the answers to the questions specified by the OTD network (Figure 4.3). The behavior policy was: step forward with $p = 0.5$, rotate right with $p = 0.25$, and rotate left with $p = 0.25$. Figure 5.3 displays the network errors averaged over 10 runs of 500,000

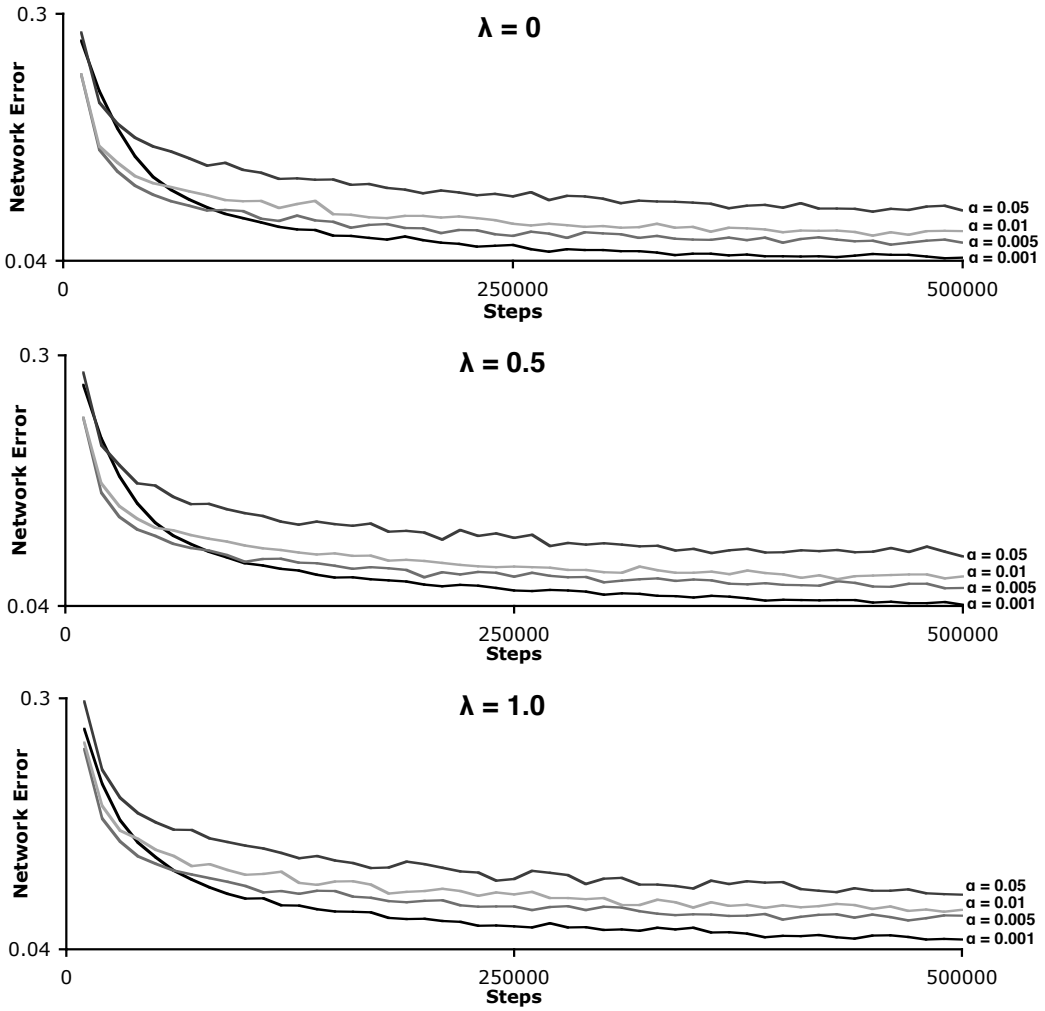


Figure 5.3: Learning curves for $\alpha = \{0.001, 0.005, 0.01, 0.05\}$ and $\lambda = \{0, 0.5, 1.0\}$. The agent reaches a lower network error for smaller values of α and λ. Not pictured are the results for $\lambda = \{0.25, 0.75, 0.9\}$ which follow the same trend. The pictured learning curves are averaged over 10 runs.

steps (network error is calculated as in Section 4.5.3). Experiments were conducted with all combinations of $\alpha = \{0.001, 0.005, 0.01, 0.05\}$ and $\lambda = \{0, 0.25, 0.5, 0.75, 0.9, 1\}$. (Results in Figure 5.3 are only displayed for $\lambda = \{0, 0.5, 1.0\}$ as they are sufficient to demonstrate the trend of the results.)

The algorithm performed best for small values of λ; after 500,000 steps the lowest average network error was achieved for every value of α with $\lambda = 0$. This is an interesting trend because it is the opposite of the results from Chapter 4 where performance improved as λ increased toward 1. A possible reason for this observed trend is that the variance of the weight updates is likely lower for smaller values of λ because the magnitude of elements in

the trace vector (whose updates are defined by Equation 5.7) were smaller as well.

Smaller values of α performed better as well, with $\alpha = 0.001$ reaching the lowest average network error for all values of λ. Other settings of α learned more quickly initially, but they were eventually surpassed by $\alpha = 0.001$. The slow start for $\alpha = 0.001$ is most likely due to the small changes that are made on each step. While agents with larger values of α can make big corrections early in training, the $\alpha = 0.001$ agent makes small changes; however, the agents with larger step-sizes are eventually unable to make appropriate fine-tuning corrections later in training, leading $\alpha = 0.001$ to perform better in the long run.

For the rest of the off-policy learning experiments presented in this chapter, the best combination of parameters among these initial experiments ($\alpha = 0.001$ and $\lambda = 0$) is used.

5.3.2 Individual Predictions

In the next set of experiments, node errors were averaged over 30 runs of 1,000,000 time steps and individual predictions were studied in further detail as shown in Figure 5.4. The four nodes presented are the error curves of the predictions for whether the orange observation bit will be active following: **Leap**, L-**Leap**, F, and **Wander**.

Most evident in Figure 5.4 is the large variance in the **Leap** and L-**Leap** predictions. There is a large performance improvement early in training (approximately the first 100,000 steps), then the error curve is quite erratic (though it decreases perceivably over time). In a limited experiment, when the algorithm was run for 10 million steps, the error did appear to continue to decrease as a trend, but there was still a large amount of fluctuation between data points. The most plausible cause of the fluctuations is the importance sampling corrections. These corrections accumulate in κ over the course of an option's execution. This explanation is consistent with the lower variance in the F and **Wander** predictions: the F node makes a one-step prediction and thus importance sampling corrections did not have the chance to grow any larger than the value of $\rho(\cdot, F)$. The low variance of the **Wander** prediction was likely due to the combination three factors:

1. There were few time steps between initiation and termination (typical execution lengths are 2 or 3 time steps).

2. The importance sampling corrections were small. The **Wander** policy was $p(\cdot, F) = \frac{1}{3}$, $p(\cdot, L) = \frac{1}{3}$, $p(\cdot, R) = \frac{1}{3}$ while the behavior policy is $p(\cdot, F) = \frac{1}{2}$, $p(\cdot, L) = \frac{1}{4}$, $p(\cdot, R) = \frac{1}{4}$, thus ρ is never larger than $\frac{4}{3}$. In conjunction with the short option executions, κ

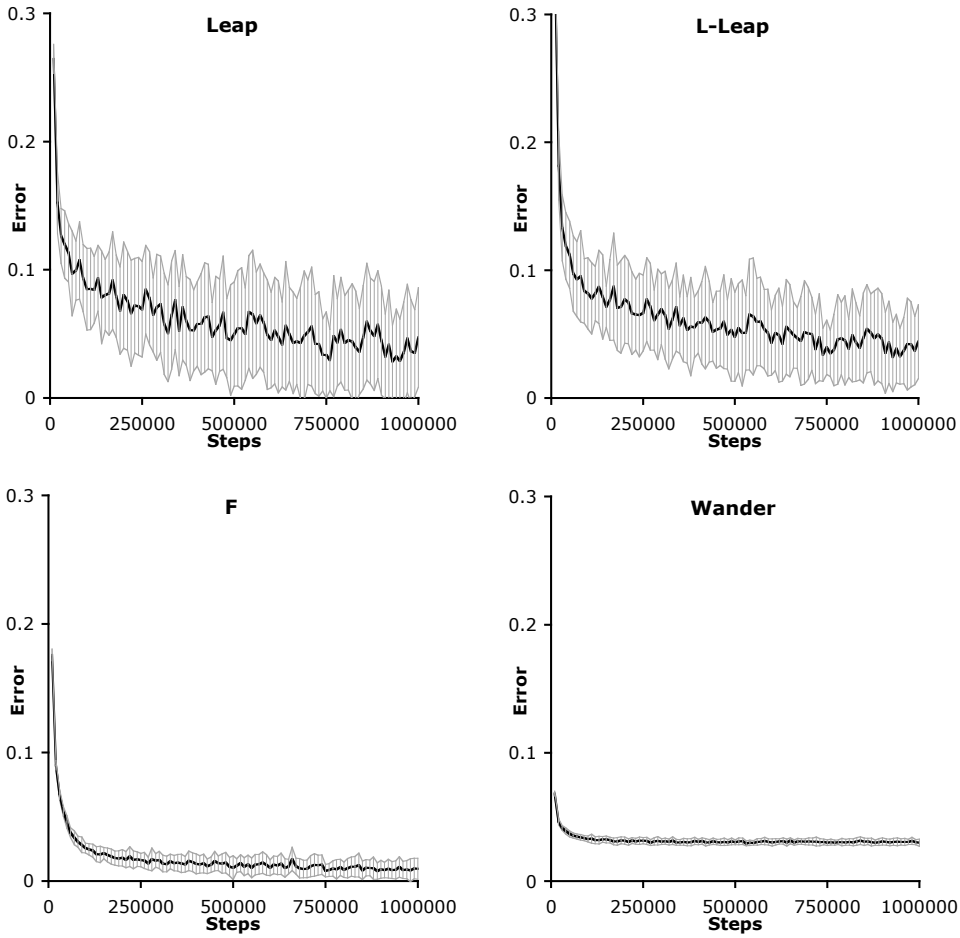


Figure 5.4: The agent was trained with the off-policy learning algorithm for 30 runs of one million time steps. The average node error is the thick line while the thin vertical lines span +/− one standard deviation. There is a large amount of variation in the error of nodes that make a **Leap** prediction.

remained small and thus the variance was small as well.

3. There was a small error on most time steps. Typically, the correct prediction (cf. Section 4.5.3) is 0 nearly everywhere. Predictions rose above 0 only when the agent was within one or two steps of the orange wall because of the short option executions. Error was consistently low over the courses of training because the predictions were initialized to 0.

5.3.3 Comparing Off-policy and On-policy Learning

The performance of the off-policy learning algorithm and the on-policy learning algorithm introduced in Section 4.4.2 were compared. Both algorithms received the 41-element feature

vector described in Section 5.3 as input. For the on-policy learning algorithm, the learning rate of $\alpha = 0.05$ (the best learning rate from the experiments in Chapter 4) was used. Because different values of λ can cause convergence to different values, the parameter λ was fixed at 0 for both algorithms.

A single training policy (and thus the same set of trajectories) was used by both the on-policy and off-policy learning agents in order to control for the effect of the policy. Actions were selected five steps in advance and on each step an oracle simulated executing the five steps. If, in the simulation, the actions caused the agent to take the **Leap** option to termination, then the on-policy agent could learn about the **Leap** option.

However, in the on-policy algorithm, the agent could only learn about one option at a time. Therefore, when the oracle indicated that the agent would execute the **Leap** option until termination, the agent chose randomly between whether it would learn about **Leap**, or it would learn about stepping forward. In this situation, the agent learned about the **Leap** option 10% of the time and learned about the prediction for stepping forward the other 90% of the time. The **Wander** node was removed from the OTD network because it was unclear how to determine whether the **Wander** option was being followed from forward simulation.

The results of the comparison between the off-policy and on-policy learning algorithms are found in Figure 5.5. The learning curves depict the average error of 30 runs of one million steps each. A random number generator (used in the action selection) was seeded to the same value for the off-policy and on-policy algorithms leading to the exact same sequences of actions being taken during the training of the off-policy agent and the on-policy agent. The results, though not entirely unexpected, were somewhat disappointing because the off-policy agent learned more slowly than the on-policy agent (indicating that the off-policy learning algorithm was less data-efficient). In terms of total network error, the on-policy algorithm learned a nearly perfect representation as its error neared 0, while the off-policy algorithm still had substantial predictive error at the end of training. Not only did the on-policy algorithm converge to a better solution, but the solution was learned more quickly than in the off-policy case.

The average error of the **Leap** predictions was studied separately with the expectation that the off-policy algorithm used data more efficiently for this prediction because the algorithm learned from both non-terminating and terminating option executions. However, the error curves for the **Leap** predictions are similar to the error curves from the entire network

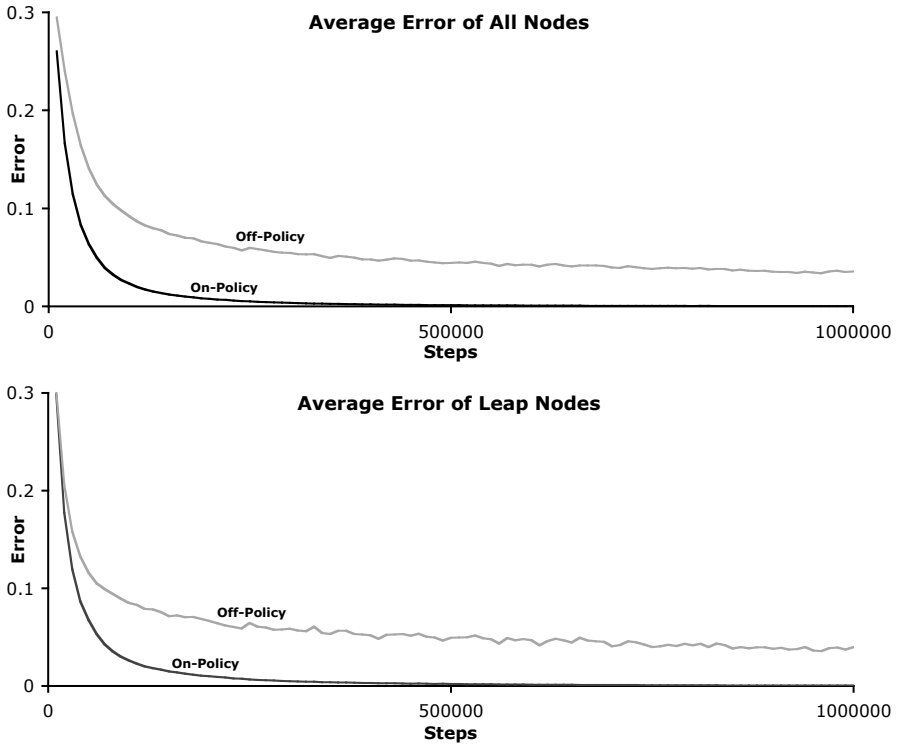
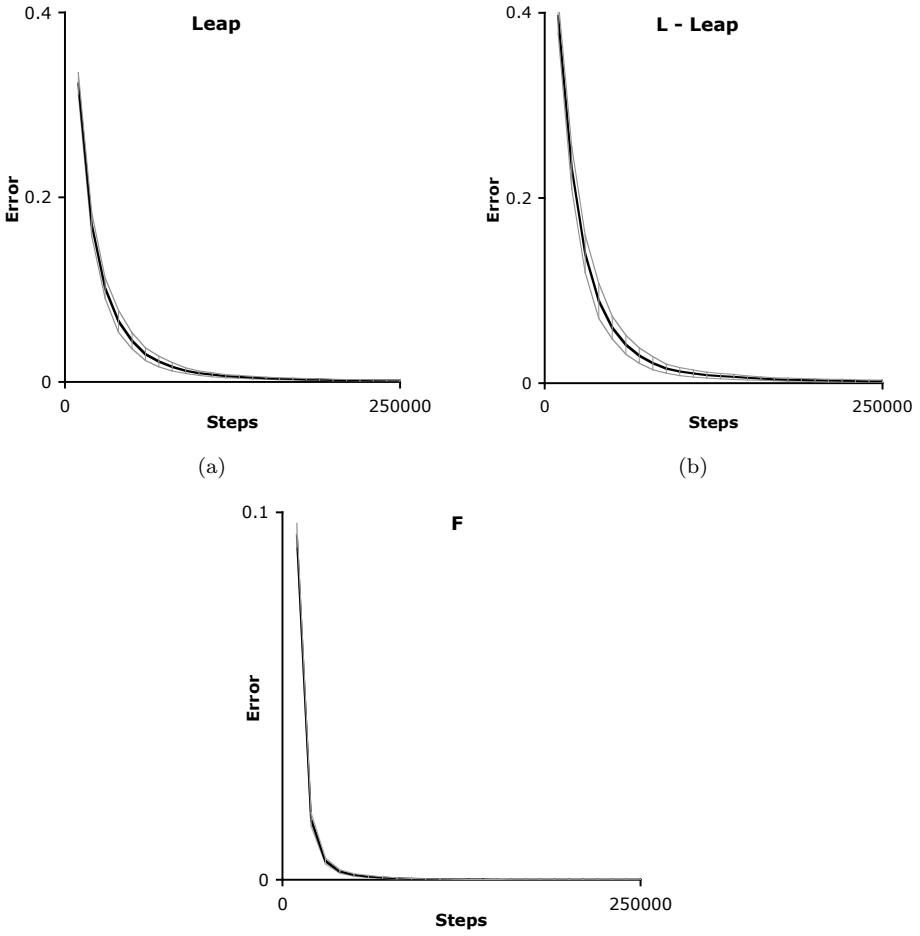


Figure 5.5: A comparison between on-policy and off-policy learning. The results are averaged over 30 runs of one million steps each. The learning rate for the off-policy algorithm was $\alpha = 0.001$ while the learning rate for the on-policy algorithm was $\alpha = 0.05$. For both algorithms, λ was fixed at 0.

(Figure 5.5). As with the average network error, the on-policy algorithm converges to a better solution and does so more quickly than the off-policy algorithm for the **Leap** nodes.

Part of this difference can be attributed to the large variance of the off-policy algorithm. Figure 5.6 displays individual node errors with error bars of one standard deviation for the on-policy algorithm. These node errors (like those presented in Figure 5.4) pertain to predictions about the activation of the orange observation bit following **Leap**, L-**Leap**, or F (the **Wander** prediction was omitted for reasons previously mentioned). Unlike the node errors of the off-policy algorithm, these node errors have low variance. There is a limited amount of variance early in learning (most visible at the elbow of the **Leap** and L-**Leap** graphs), but the variance quickly becomes negligible as the prediction error drops to 0. The variance in the F prediction is so small that the error bars are nearly imperceptible at any point in the graph. While the on-policy agent learns consistently, there is large variance between the quality of the models learned by the off-policy agent.



(a) (b)

(c)

Figure 5.6: Individual Node errors for the on-policy algorithm with error bars of one standard deviation. In all cases variance decreases over time and eventually becomes almost imperceptible. Notice that these graphs stop at time step 250,000. Notice also that the x-axis in (c) is of a smaller scale than in (a) and (b).

5.4 Discussion and Conclusions

This chapter introduced the first algorithm for the off-policy learning of option models and proved it to have the same expected updates as the on-policy learning algorithm. The algorithm was obtained by deriving an incremental update rule from a forward-view algorithm. Importance-sampling corrections were introduced to account for the difference between the behavior policy and the target policy. Experimental results demonstrated that prediction error in the grid world originally introduced in Chapter 4 decreases over time.

However, the product of importance-sampling corrections may become large, leading to a large amount of variance in the updates—the prediction error fluctuated over the course of

multiple training runs. The off-policy learning algorithm performed poorly in comparison to the on-policy algorithm presented in Chapter 4, even when both algorithms were provided with the same experience.

The problems with variance are by no means a reason to discount off-policy learning. If the off-policy algorithm can be extended to control the growth of the importance-sampling corrections, and thus reduce the magnitude of weight updates, then data could perhaps be used more efficiently (possibly more so than the on-policy algorithm). Controlling the empirical variance would result in an improved learning rate for the off-policy algorithm, and convergence would still be guaranteed. A possible extension to the off-policy algorithm is the implementation of recognizers (Precup et al., 2005) which have been shown to reduce the variance of importance-sampling corrections.

Despite the negative results encountered in the experiments, off-policy learning is important because an agent receives only a single stream of data and this data must be used to learn as much as possible about the environment. As options take longer to terminate, the probability of executing an option until termination decreases. An agent that learns off-policy can learn from these non-terminating trajectories whereas an on-policy agent would learn nothing. Off-policy learning is a critical issue for learning agents and requires further study.

Chapter 6

Putting It All Together

This chapter investigates the intersection of the topics covered in the previous two chapters: learning an OTD network and learning off-policy. The off-policy learning of a TD network is complicated by the non-stationary nature of the feature vector. This non-stationarity stems from the fact that the agent is attempting to learn its own state representation. In the previous chapter the agent's observation was assumed to be a feature vector drawn from a stationary distribution. But this assumption is not valid when the feature vector is constructed from learned predictions. The predictive feature-vector at an environmental state may vary depending on how much experience the agent has in the world, how the agent arrived at the state, and how the learning parameters have been initialized. The predictive state is meant to be a sufficient statistic once learned, but throughout the learning process can be potentially inaccurate. In this chapter, empirical results suggest that an off-policy agent can learn the predictions specified by an OTD network despite the lack of theoretical guarantees.

6.1 Learning OTD Networks Off-policy

Chapter 4 presented an on-policy algorithm learning for option-conditional TD networks; Chapter 5 presented an off-policy algorithm for learning option models. These two algorithms are combined into the first off-policy algorithm for learning OTD networks.

The off-policy learning algorithm from the previous chapter only needs a minor adjustment in order to be applied to OTD network learning. In particular, the feature vector $\boldsymbol{\phi}_t$ in Chapter 5 was produced as a function of the agent's environmental state. In this chapter, $\boldsymbol{\phi}_t$ is constructed as in Equation 4.5; that is, $\boldsymbol{\phi}_t$ is constructed from the agent's predictions, $\mathbf{y}_{t-1}$, the last action taken, a_{t-1}, and the current observation o_t. In general, $\boldsymbol{\phi}_t$ can be

constructed from any arbitrary function over these values.

While a linear function approximator is typically used to compute the predictions, a common procedure in the field of machine learning is to use a set of non-linear features in the linear approximator (e.g., the action-conditional approach described in Section 4.5.2). Action-conditional feature-vector construction is but one approach to adding non-linear features to a TD network; other conceivable approaches are to take the logical AND or the logical OR of predictions or to use a thresholding function to discriminate between values of a continuous-valued feature. These approaches are merely suggestions and many other approaches to feature construction exist.

The off-policy algorithm for learning OTD networks is a combination of the previous algorithms and computes values in the following order:

1. κ_t update: Equation 5.6
2. Trace ($\mathbf{E}_t$) update: Equation 5.7
3. $\boldsymbol{\phi}_{t+1}$ update: Equation 4.5
4. $\tilde{\mathbf{y}}_{t+1}$ update: Equation 4.20
5. δ_t update: Equation 5.10
6. Weight ($\boldsymbol{\theta}_{t+1}$) update: Equation 5.11
7. Prediction ($\mathbf{y}_{t+1}$) update: Equation 4.3

Pseudocode for implementing the algorithm can be found in Algorithm 3.

6.2 Experiments

As in the previous two chapters, this chapter's learning algorithm was tested in the colored grid-world (Figure 4.2) using the 45 node question network illustrated in Figure 4.3 along with action-conditional feature-vector construction (as described in Section 4.5.2).

6.2.1 Parameter Study

Figure 6.1 shows an initial examination of the learning parameters: all combinations of $\alpha = \{0.0005, 0.001, 0.005, 0.01\}$ and $\lambda = \{0, 0.5, 1.0\}$. The results were averaged over 10 runs of 500,000 steps for each parameter combination. The vertical axis of the graphs represents the network error (cf. Section 4.5.3). In all cases the error descended over time.

Algorithm 3 The off-policy OTD network algorithm.

Initialize $\mathbf{y}_0, \mathbf{E}_0, \boldsymbol{\theta}_0, \boldsymbol{\beta}_0, \boldsymbol{\kappa}_0, \boldsymbol{\rho}_0$

for $t = 1, 2, \ldots$

  Take action a_t; receive feature vector $\boldsymbol{\phi}_{t+1}$

  Update product of importance sampling corrections: $\boldsymbol{\kappa}_t = \boldsymbol{\rho}_{t-1}\boldsymbol{\kappa}_{t-1}(1 - \beta_t) + \mathbf{g}_t$

  Compute importance sampling corrections: $\boldsymbol{\rho}_t = \boldsymbol{\rho}(a_t)$

  Update trace matrix: $\mathbf{E}_t = \lambda(1 - \boldsymbol{\beta}_t)\boldsymbol{\rho}_{t-1}\mathbf{E}_{t-1} + \boldsymbol{\kappa}_t \nabla_{\boldsymbol{\theta}_t} \mathbf{y}_t$

  Construct feature vector: $\boldsymbol{\phi}_{t+1} = \boldsymbol{\phi}(\mathbf{y}_t, a_t, o_{t+1})$

  Compute interim predictions: $\tilde{\mathbf{y}}_{t+1} = \mathbf{u}(\boldsymbol{\phi}_{t+1}, \boldsymbol{\theta}_t)$

  Check for termination: $\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}(o_{t+1}, \mathbf{y}_t)$

  Update target values: $\mathbf{z}_t = \mathbf{z}(o_{t+1}, \tilde{\mathbf{y}}_{t+1})$

  Compute error: $\boldsymbol{\delta}_t = \boldsymbol{\rho}(\boldsymbol{\beta}_{t+1}\mathbf{z}_{t+1} + (1 - \boldsymbol{\beta}_{t+1})\tilde{\mathbf{y}}_{t+1}) - \mathbf{y}_t$

    (multiplications are component-wise)

  Update weights: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\boldsymbol{\delta}_t\mathbf{E}_t$

  Update predictions: $\mathbf{y}_{t+1} = \mathbf{u}(\boldsymbol{\phi}_{t+1}, \boldsymbol{\theta}_{t+1})$

end for

The lowest errors were achieved when $\alpha = 0.001$ (as in the parameter study of Section 5.3.1). Error decreased most rapidly for $\lambda = 1$, but all three settings of λ resulted in similar error values after 500,000 steps. Only for $\alpha = 0.01$ was there a clear trend visible among the values of λ (error decreases as λ increases). For $\alpha = 0.001$, it was unclear whether $\lambda = 0$ or $\lambda = 1.0$ is the better parameter setting. While $\lambda = 1.0$ learns more quickly than $\lambda = 0$, our results from the previous chapter suggest that a higher value of λ is related to higher variance. Further experiments (Figure 6.2) helped distinguish between the two settings of λ.

The learning curves in Figure 6.2 are the result of 30 runs of one million steps each. The solid black line is the network error, averaged over the 30 runs, and the grey lines show $+/-$ one standard deviation. As expected, the variance was lower when $\lambda = 0$. The weight updates were smaller for $\lambda = 0$ because the trace update equation for $\lambda = 0$ (cf. Equation 5.7) only contains a single-step trace (and not a trace over the entire trajectory). A lower error was also achieved for $\lambda = 0$, though the difference was not statistically significant.

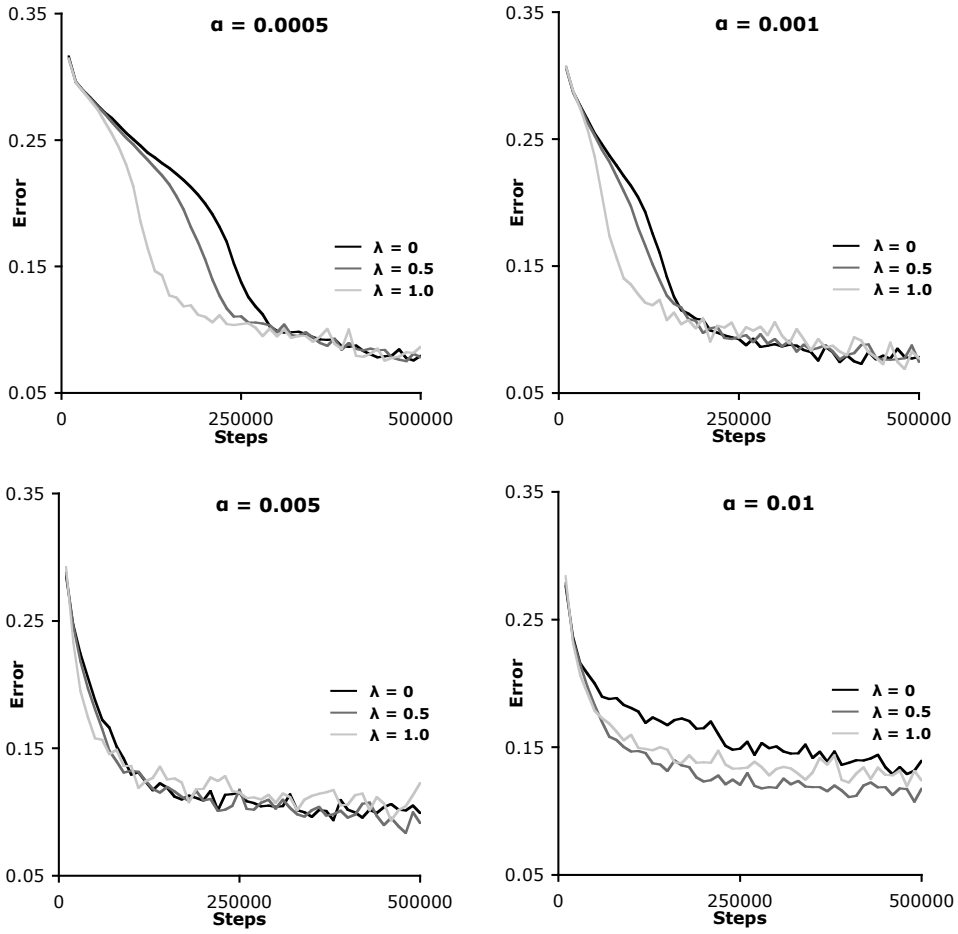


Figure 6.1: Learning curves for various combinations of $\alpha = \{0.0005, 0.001, 0.005, 0.01\}$ and $\lambda = \{0, 0.5, 1.0\}$. These curves were generated by running the off-policy OTD network for 10 runs of 500,000 steps each. The best learning rate is generally achieved when $\lambda = 1$ and the errors are lowest for $\alpha = 0.001$. It is difficult to distinguish whether any value of λ leads to a better solution after 500,000 steps because there is a large amount of fluctuation in the average error.

6.2.2 The Concept of Direction, Revisited

Section 4.5.6 presented an example of a trained agent that could keep track of its direction for an indefinite amount of time, computing its next set of predictions from current predictions. The agent was trained on policy, and when manually steered through the environment, was demonstrated to make the correct predictions (cf. Figure 4.8).

As a demonstration of the correctness of the off-policy learning algorithm, the agent was steered through Section 4.5.6's 29-step sequence of actions (six complete clockwise rotations, three steps forward and a counter-clockwise rotation). The predictions made by the manually controlled agent after 1 million training steps are shown in Figure 6.3.

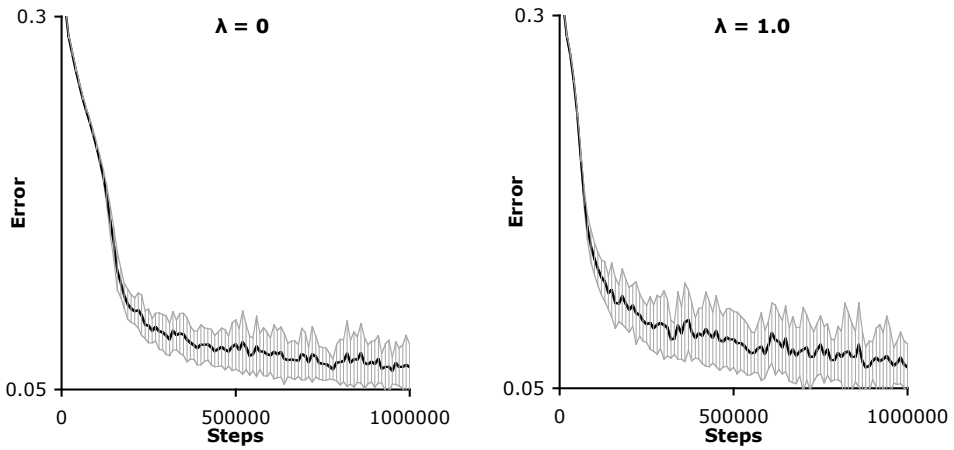


Figure 6.2: Learning curves for $\lambda = 0$ and $\lambda = 1$ (both with $\alpha = 0.001$). The graphs show the node error averaged over 30 runs of one million steps (solid black line) and one standard deviation in either direction (grey lines). $\lambda = 0$ has both a lower variance and a lower final error.

As before, the agent correctly maintains direction, as seen by the predictions for the **Leap** node. The agent keeps track of which color it would see if it were to step forward until reaching a wall. However, on time step 4, there is a difference between the on-policy and off-policy agents' **Leap** predictions. The on-policy agent predicted blue with a probability of roughly $\frac{5}{6}$ and green with a probability of roughly $\frac{1}{6}$. The off-policy agent no longer predicted seeing green, but still predicted seeing blue with a probability still roughly $\frac{5}{6}$. This may have been the result of the sequence of actions prior to reaching the state at $t = 1$. The agent's predictions were reverted to those at $t = 1$ and the agent was manually stepped forward once, and rotated left once. The prediction for **Leap** was then the same $\frac{5}{6}$:$\frac{1}{6}$ ratio observed in the on-policy experiments. The action-conditional feature-vector construction causes the past action to impact the computation of the agent's predictions and therefore it is possible that the absence of a green prediction is a result of the preceeding rotate-right action.

A second strange result was the presence of a prediction for blue if the sequence **Leap**-L-**Leap** is followed when the agent is facing North (time steps 1, 5, and 25). This sequence will always navigate the agent into the upper left corner of the world, and upon reaching the corner the agent will always be facing the green cell (the agent should thus only predict green). It is possible that the prediction of blue was a result of learning from non-terminating executions of the **Leap** option (the agent initiated the **Leap** option, but diverged before

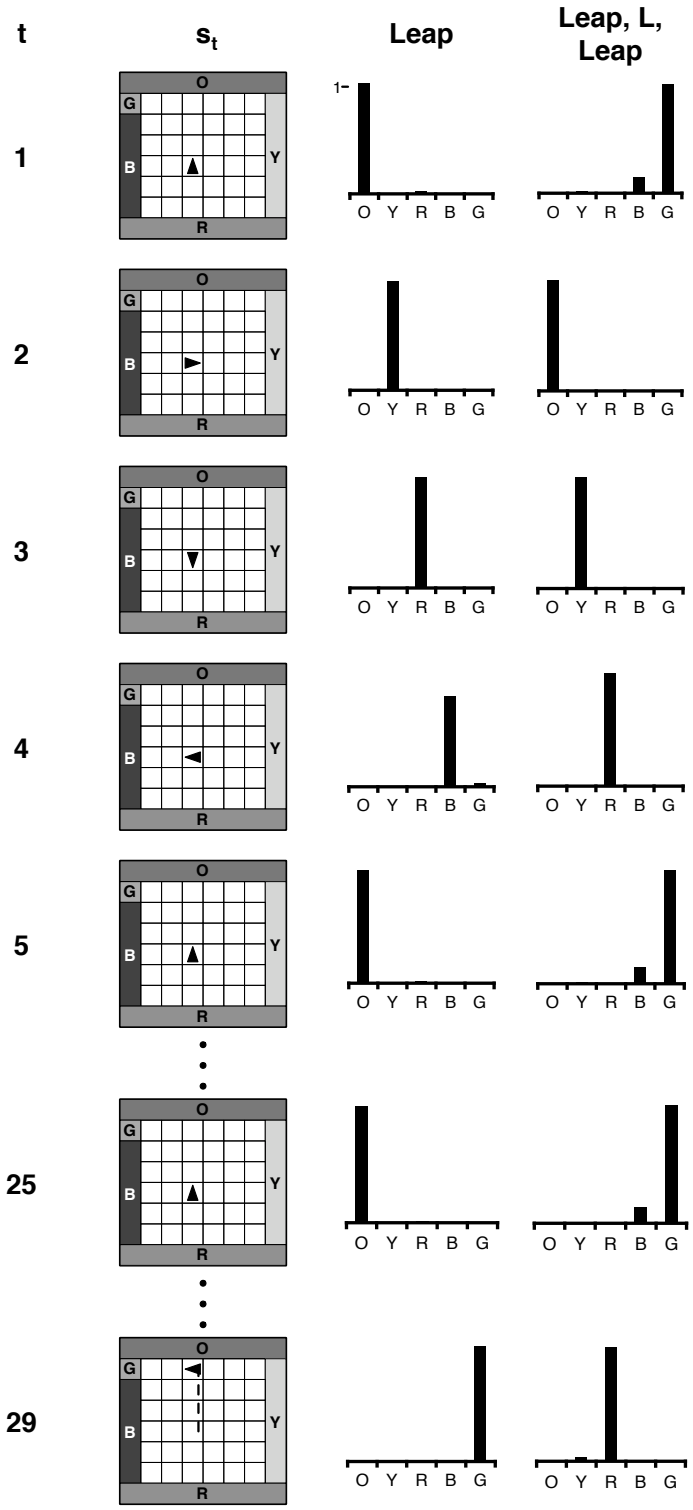


Figure 6.3: The agent learns a compass with the off-policy learning algorithm. After one million steps of training, the agent was manually guided through a 29-step sequence of actions, recording the predictions made at each time step (the same sequence of actions taken in the on-policy experiment in Figure 4.5.6) The first column contains the relative time index, the second column indicates the agent's position in the world, and the last two columns indicate the value of the **Leap** and L-**Leap** predictions for each of the five color-observation bits.

the termination condition was satisfied). When facing West, the target for **Leap** can be either blue or green depending on the agent's row. The prediction for L-**Leap** targets the **Leap** prediction and thus L-**Leap** may also predict blue or green. In turn, the prediction for **Leap**-L-**Leap** targets the L-**Leap** prediction, and because the first **Leap** option in the sequence may not always be followed until termination the targeted L-**Leap** prediction can be either blue or green, leading the **Leap**-L-**Leap** node to predict both blue and green.

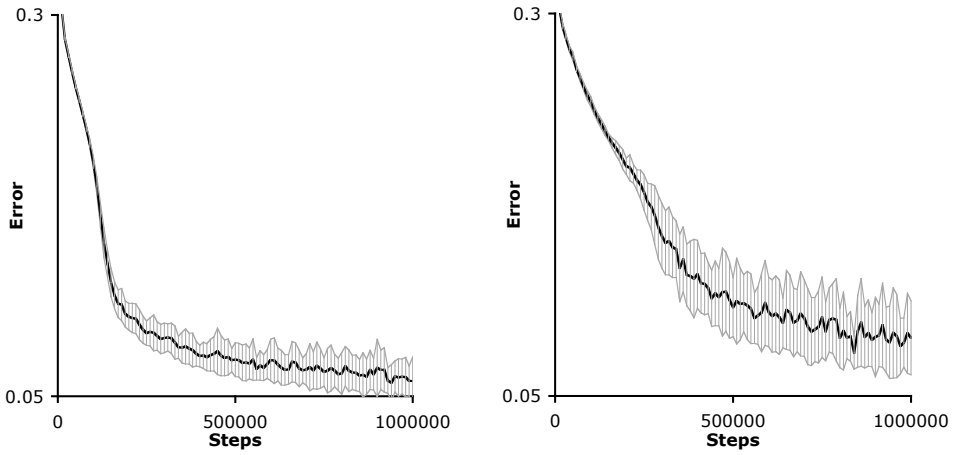
In general, the predictions are very similar between the on-policy and off-policy algorithms and the agent clearly demonstrates the ability to track its direction when trained with the off-policy algorithm. After training, an agent could spin in the middle of the environment for an arbitrary amount of time, all the while tracking its current direction. The concept of direction is learned despite the fact that the agent is learning off-policy and never explicitly chooses to follow any of the options that it is learning about. The current direction (and more generally, all current predictions) is (are) maintained as a function of previous predictions, all of which are learned from experience in the world.

6.2.3 Different Behavior Policies

The final experiment in this chapter tested the off-policy learning algorithm with behavior policies other than $b_0 = \{p(\cdot, F) = \frac{1}{2},\ p(\cdot, R) = \frac{1}{4},\ p(\cdot, L) = \frac{1}{4}\}$, which was used in all off-policy experiments so far (learning curves for an agent trained with b_0 are shown in Figure 6.2). Figure 6.4 presents the network error and errors bars as shown for two new policies. In these experiments $\alpha = 0.001$ and $\lambda = 0$. The error curves for the new behavior policies $b_1 = \{p(\cdot, F) = 0.55,\ p(\cdot, R) = 0.3,\ p(\cdot, L) = 0.15\}$ and $b_2 = \{p(\cdot, F) = \frac{1}{3},\ p(\cdot, R) = \frac{1}{3},\ p(\cdot, L) = \frac{1}{3}\}$ are shown in Figures 6.4a and 6.4b.

With all three behavior policies, the error dropped quickly at first before learning slowed, but continued to improve steadily. The lowest error and lowest variance occurred with b_1 as the behavior policy, a policy where an imbalance existed between the probabilities of rotating right and rotating left. There are large fluctuations between data points along the error curve for b_2; in comparison, the error curve for b_1 is much smoother.

Part of the reason for the algorithm's poorer performance for b_2 may be that most actions were rotations—the agent had a lower probability of executing the **Leap** option until termination and thus received less data about terminating sequences. Depending on the initial state, an agent required up to five step-forward actions to execute the **Leap** option until termination. The probability of executing **Leap** under b_0 was thus as low as



(a) b_1 (b) b_2

Figure 6.4: Experiments with the behavior policies b_1 and b_2 (see text). The results are averaged over 30 runs of one million steps each. The thick line is the average error of the 30 runs while the grey lines mark $+/-$ one standard deviation.

$(P_{b_0}(\cdot, F))^5 = (\frac{1}{2})^5 = 0.03125$; the probability of executing **Leap** under b_2 was as low as $(P_{b_1}(\cdot, F))^5 = (\frac{1}{3})^5 = 0.00412$ (almost an order of magnitude lower than b_0).

The b_0 policy was originally chosen to both promote the execution of the **Leap** option until termination and promote the exploration of the the grid world's interior cells. b_2, the uniform random policy, may cause the agent to spend most of its time in the middle of the grid world where very little can be learned.

Figure 6.5 shows a comparison between the error curves of the L-**Leap** and R-**Leap** nodes for the policies b_0 and b_1. This experiment investigates whether the imbalance between rotate right and rotate left had any effect on learning. In both cases, the learning rates and errors are approximately the same for both behavior policies.

6.3 Discussion and Conclusions

In this chapter, the off-policy learning algorithm derived in Chapter 5 was applied to OTD network learning. Experimental results indicated that the agent could learn a model of the previously introduced colored grid-world. The difference between the algorithm presented in this chapter and the algorithm from Chapter 5 is that the predictions generated by the OTD network were used as features for the new algorithm. Though using predictions as features causes features to be non-stationary, the agent still learned a set of weights that enabled it to make accurate predictions. The agent was also demonstrated to maintain the

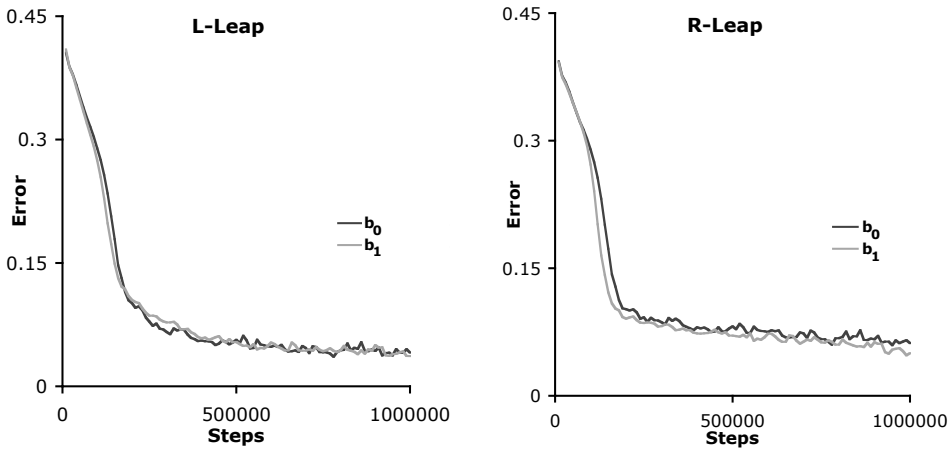


Figure 6.5: Learning rates for the L-**Leap** and the R-**Leap** nodes using the behavior policies b_0 and b_1 (see text) was compared. The error curves are very similar for both behavior policies, indicating that importance sampling corrects for the difference between policies.

same concept of direction as the on-policy agent from Chapter 4. The agent also learned predictions when trained with three different behavior policies

However, experiments were not conducted for behavior policies with extreme action selection probabilities (very small or very large probabilities of selecting certain actions) because this would lead to large importance-sampling corrections and thus exacerbate the variance of weight updates. In theory, even the most extreme importance-sampling corrections would not be problematic given a small enough step-size and an infinite amount of training, but in practice, the amount of training the agent can receive is bounded and determining an appropriate step-size may be a tedious process. A more sophisticated algorithm that can automatically tune step-sizes or bound the magnitude of importance-sampling corrections could potentially control for extreme action-selection probabilities.

In the experiment in Section 6.2.2 the absence of certain predictions and the presence of others was especially conspicuous. Possible reasons were suggested to account for the errors, but another reason, tied to the issue of variance, is also plausible. Due to the variance between training runs, it was possible that the prediction error was an artifact of the specific training run. The agent was retrained on one million different steps of data and the prediction error for the **Leap**-L-**Leap** node was no longer present. Again, whether this second training run is representative of learning comes into question, but the difference between training runs clearly demonstrates the off-policy algorithm's problem with variance.

Chapter 7

Conclusion

The primary contribution of this thesis is the OTD-network learning algorithm that included temporal abstraction in TD networks by incorporating the options framework. The inclusion of options allowed high-level, temporally-abstract concepts to be learned from data (actions and observations). Introduced in this thesis were algorithms for:

- the on-policy learning of option-conditional temporal-difference (OTD) networks (Chapter 4)
- the off-policy learning of option models (Chapter 5)
- the off-policy learning of OTD networks (Chapter 6)

The problem of off-policy learning, in which an agent learns about multiple options from a single stream of data, was studied in detail in Chapters 5 and 6. The algorithms introduced in these chapters incorporated importance-sampling corrections. Another contribution of this work was a test of the *predictive representations hypothesis* in which TD networks were demonstrated to perform useful state abstraction (Chapter 3).

7.1 Future Work

As discussed in Chapter 1, *what can be represented* and *how a representation is learned* are studied in this thesis. In addition to representation and learning, I believe that there is a third issue (not treated in this thesis)—*discovery*—that deserves consideration. Discovery can be described as *learning what to learn*. In a TD network or an OTD network, the problem of discovery is the problem of learning the structure of the question network. Future research on the topics of representation, learning, and discovery (and how the issues are interrelated) are discussed in this section.

7.1.1 Representation

Chapter 3 explored the representational power of a TD network. In the experiments, a reinforcement learning agent's state representation was constructed from a TD network's predictions. The results of learning were promising and deserved further exploration.

Figure 3.1 discussed the confounding factors and the corresponding steps taken to control for them. Future experiments could be broadened in scope and allow the presence of certain confounding factors. For example, experiments could be conducted in stochastic environments. However, an oracle would not predict binary values in these environments. Instead, there may be a probability associated with receiving an observation at the end of a test. The predictions could be represented in a tabular form or could be used as inputs for a function approximator. Tabular predictive classes could be constructed from continuous-valued predictions by defining a soft notion of equivalence—configurations within some distance of each other (according to some metric) could be grouped into a class. On the other hand, continuous-valued predictions could be used directly as features (reinforcement-learning algorithms, such as Sarsa(λ) and Q-Learning, can be used with function approximation (Sutton & Barto, 1998)).

Removing the need for oracle-generated predictions is also a potential direction for this research—combining the study of representation and learning. Experiments could be designed to test the simultaneous learning of the predictions and learning of a solution to a reinforcement learning problem. Concurrently learning both a set of predictions and a solution to a task is difficult because the predictions serve as the agent's state. Because the predictions are learned, the state representation is constantly changing, potentially causing problems with the learning task.

7.1.2 Learning

Section 3.3.2 identified a trade between asymptotic performance and speed of learning, but must it be a trade? Are the two mutually exclusive? Both issues, as they apply to TD network learning, are worthy of further study and the hope is that an algorithm exists that learns predictions accurately, and learns them quickly.

Controlling Variance

As seen in Chapters 5 and 6, a problem existed with the variance between training runs because of the magnitude of weight updates. A proposed reason for the large weight changes

is the exponential growth of κ, the product of importance sampling corrections.

The experiments in Chapters 5 and 6 attempted to deal with potentially large values of κ_t by using a small step-size; however, a better solution may be to incorporate an approach that adapts step-sizes over the course of training (Sutton, 1992). Another possible solution to the problem of large variance is to simply bound κ, effectively bounding the magnitude of weight changes. Rather than bounding κ, another possible approach is to control its magnitude by dividing by κ_{max}, the largest value of κ encountered during training. Finally, the recognizer framework introduced by Precup et al. (2005) touches on reducing the variance of off-policy learning. Defining option policies as recognizers may result in lower variance in the weight updates.

An Empirical Demonstration of Weight Update Equivalence

An interesting result would be a thorough empirical test of Equation 5.14 which states that over the course of an option, the on-policy algorithm and off-policy algorithms have the same expected updates. This equation holds when the agent's observation is a stationary feature vector (as in Chapter 5).

Section 5.2.4 discussed a small experiment for which Equation 5.14 held; however, the experimental world was deterministic and the target policy chose the same action on every time step (**Leap**). A more thorough set of experimental tests could help to suggest that the theoretical result holds in practice. These tests could be conducted in a world with stochastic transitions and the agent would learn about a more sophisticated option (such as **Wander**).

Fast Learning

In our experiments, agents generally received several hundred thousand steps of training. To make the presented algorithms attractive for use with real-world data, algorithms need to use data more efficiently because real-world data is often more expensive to acquire.

Off-policy learning is meant to help accelerate learning, but there are also other methods for improving learning rates. Tanner and Sutton introduced a TD-network learning algorithm with inter-node traces which greatly reduced the amount of data needed to learn environments (2005). Inter-node traces could likely be incorporated in the OTD network algorithm as well.

Another approach to accelerating learning is the possible decorrelation of inputs. There

may be redundant nodes in a TD network which, if detected, could be removed, reducing the size of the feature vector and thus helping to accelerate learning. It may be possible to conduct something like principal component analysis on the predictions to reduce the dimensionality of the representation.

Finally, the agent was always trained with a random policy in this work: the on-policy agent selected randomly between options, and the off-policy agent selected randomly between simple actions. A *directed exploration* strategy would likely improve the agent's learning speed—behavior during training could be chosen to accomplish goals tied to knowledge acquisition. Rather than choosing actions and options randomly, the agent could choose a behavior policy that would, for example, explore unknown regions of the state space, or constrain the agent to a region of the state space until predictions were made perfectly in the region.

7.1.3 Discovery

An important step forward for temporal-difference networks is the development of a discovery algorithm. Currently, question networks are specified in advance (for both TD networks and OTD networks), but an agent would ideally add and remove predictions from the question network over the course of learning.

Predictions could be added in many ways. In a TD network, a simple discovery algorithm could be developed by incrementally increasing the number of levels in the question network until the predictions are a sufficient statistic (reminiscent of James and Singh's PSR discovery algorithm (2004)). Another possibility is a generate-and-test approach in which a new prediction is added, then after some training the prediction is retained only if it provides useful information (prediction has low error, the inclusion of the prediction decreases total network error, etc.). A genetic algorithm could even be used to address the discovery problem. Multiple TD networks with randomly generated question networks could be trained, and the question networks would be combined based on fitness (network error), then training would start anew.

Discovery in an OTD network could be conducted by option "sculpting"—beginning with a very general option (perhaps similar to the **Wander** option), a specific prediction could be made by modifying the option over the course of training. This process could begin by identifying a desired outcome then learning a policy and termination condition for which the desired outcome is likely to be observed.

The discovery problem for TD networks and OTD networks is largely unexplored, any progress along this line of research would be welcome.

7.2 Discussion

This thesis is a small step towards addressing a grand challenge of knowledge representation: learning high-level concepts from low-level observables. Connecting concepts with data is critical in the development of autonomous systems because knowledge is represented in a form that is accessible to the agent. Predictive representations, and more specifically option-conditional temporal-difference networks, address the grand challenge by learning models in which predictions are related to concepts.

A key result of the experiments presented in this thesis is the emergence of the concept of direction. After training, an agent moved into the middle of space kept track of its direction when spun in circles. This is an important result because the concept is not constrained by history—the agent can be spun for an indefinite amount of time and the agent will never lose track of direction. Also, as the agent is spun, the agent's observation provides no directional information, indicating that the agent is updating its predictions from previous predictions.

Abstraction, over both state and time, is what allows the concepts to be represented. Spatial abstraction generalizes the environmental state by grouping situations with similar sets of predictions; temporal abstraction allows sequences of actions to be treated as single units. The concept of direction involves both types of abstraction: the agent knows the direction it is facing because it can predict the outcome of a temporally-abstract sequence of actions, and this prediction is computed from features of its current abstract state (the current set of predictions).

Steps made towards developing an off-policy learning algorithm are also significant. In the real-world there is no such thing as multiple runs, there is only a single stream of experience and all learning stems from this experience. This demonstrates the need for off-policy learning—there are many outcomes to learn about, but only a single stream of data. The off-policy algorithm successfully learns the outcomes of temporally-extended behaviors in two cases: when a set of features are observed by the agent, and perhaps more interestingly when features are constructed from predictions. It is in the second case that the agent benefits from both abstraction and off-policy learning, and it is this case that will allow agents to model larger and more complex worlds.

Bibliography

[Baird, 1995] Baird, L. C. (1995). Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 30–37.

[Bertsekas and Tsitsiklis, 1996] Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-dynamic Programming*. Athena Scientific, Belmont, MA.

[Bowling et al., 2006] Bowling, M., McCracken, P., James, M., Neufeld, J., and Wilkinson, D. (2006). Learning predictive state representations using non-blind policies. In *Proceedings of the Twenty-Third International Conference on Machine Learning (ICML)*, pages 129–136.

[Cassandra et al., 1997] Cassandra, A., Littman, M. L., and Zhang, N. L. (1997). Incremental pruning: A simple, fast, exact algorithm for partially observable Markov decision processes. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence*, pages 54–61.

[Crites and Barto, 1996] Crites, R. H. and Barto, A. G. (1996). Improving elevator performance using reinforcement learning. In *Advances in Neural Information Processing Systems 8*, pages 1017–1023.

[Dietterich, 1998] Dietterich, T. G. (1998). The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 118–126.

[Drescher, 1991] Drescher, G. (1991). *Made-up Minds: A Constructivist Approach to Artificial Intelligence*. MIT Press.

[Holmes and Isbell Jr., 2004] Holmes, M. P. and Isbell Jr., C. L. (2004). Schema learning: Experience-based construction of predictive action models. In *Advances in Neural Information Processing Systems 17*, pages 585–592.

[Hundt et al., 2006] Hundt, C., Panagaden, P., Pineau, J., and Precup, D. (2006). Representing systems with hidden state. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 368–374.

[Jaeger, 1998] Jaeger, H. (1998). A short introduction to observable operator models for stochastic processes. In *Proceedings of the 1998 Cybernetics and Systems conference*, volume 1, pages 38–43.

[Jaeger, 2000] Jaeger, H. (2000). Observable operator models for discrete stochastic time series. *Neural Computation*, 12(6):1371–1398.

[James and Singh, 2004] James, M. R. and Singh, S. (2004). Learning and discovery of predictive state representations in dynamical systems with reset. In *Proceedings of the Twenty-First International Conference on Machine Learning (ICML)*, pages 417–424.

[James and Singh, 2005] James, M. R. and Singh, S. (2005). Planning in models that combine memory with predictive representations of state. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI)*, pages 987–992.

[James et al., 2004] James, M. R., Singh, S., and Littman, M. L. (2004). Planning with predictive state representations. In *Proceedings of the 2004 International Conference on Machine Learning and Applications (ICMLA)*, pages 304–311.

[James et al., 2005] James, M. R., Wolfe, B., and Singh, S. (2005). Combining memory and landmarks with predictive state representations. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 734–739.

[Littman et al., 2002] Littman, M. L., Sutton, R. S., and Singh, S. (2002). Predictive representations of state. In *Advances in Neural Information Processing Systems 14*, pages 1555–1561. MIT Press.

[McCallum, 1996] McCallum, A. K. (1996). *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, Department of Computer Science, University of Rochester, Rochester, New York.

[McCracken and Bowling, 2005] McCracken, P. and Bowling, M. (2005). Online discovery and learning of predictive state representation. In *Advances in Neural Information Processing Systems 18*, pages 875–882.

[Mitchell, 2003] Mitchell, M. W. (2003). Using Markov-k memory for problems with hidden-state. In *MLMTA*, pages 242–248. CSREA Press.

[Precup et al., 2001] Precup, D., Sutton, R. S., and Dasgupta, S. (2001). Off-policy temporal-difference learning with function approximation. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 417–424.

[Precup et al., 2005] Precup, D., Sutton, R. S., Paduraru, C., Koop, A. J., and Singh, S. (2005). Off-policy learning with recognizers. In *Advances in Neural Information Processing Systems 18*, pages 1097–1104.

[Precup et al., 2000] Precup, D., Sutton, R. S., and Singh, S. (2000). Eligibility traces for off-policy evaluation. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 759–766.

[Rafols et al., 2005] Rafols, E. J., Ring, M. B., Sutton, R. S., and Tanner, B. (2005). Using predictive representations to improve generalization in reinforcement learning. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 835–840.

[Ring, 1994] Ring, M. B. (1994). *Continual Learning in Reinforcement Environments*. PhD thesis, University of Texas at Austin, Austin, Texas 78712.

[Rivest and Schapire, 1994] Rivest, R. L. and Schapire, R. E. (1994). Diversity-based inference of finite automata. *J. ACM*, 41(3):555–589.

[Rosencrantz et al., 2004] Rosencrantz, M., Gordon, G., and Thrun, S. (2004). Learning low dimensional predictive representations. In *Proceedings of the Twenty-First International Conference on Machine Learning (ICML)*, pages 88–95.

[Rudary and Singh, 2004] Rudary, M. R. and Singh, S. (2004). A nonlinear predictive state representation. In *Advances in Neural Information Processing Systems 16*, pages 855–862.

[Rudary and Singh, 2006] Rudary, M. R. and Singh, S. (2006). Predictive linear-gaussian models of controlled stochastic dynamical systems. In *Proceedings of the Twenty-Third International Conference on Machine Learning (ICML)*, pages 777–784.

[Rudary et al., 2005] Rudary, M. R., Singh, S., and Wingate, D. (2005). Predictive linear-gaussian models of stochastic dynamical systems. In *Uncertainty in Artificial Intelligence: Proceedings of the Twenty-First Conference*, pages 777 – 784.

[Singh et al., 2004] Singh, S., James, M. R., and Rudary, M. R. (2004). Predictive state representations: A new theory for modeling dynamical systems. In *Uncertainty in Artificial Intelligence: Proceedings of the Twentieth Conference*, pages 512–519.

[Singh et al., 2003] Singh, S., Littman, M., Jong, N., Pardoe, D., and Stone, P. (2003). Learning predictive state representations. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML)*, pages 712–719.

[Sutton, 1988] Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44.

[Sutton, 1992] Sutton, R. S. (1992). Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *The Tenth National Conference on Artificial INtelligence*, pages 171–176.

[Sutton, 1995] Sutton, R. S. (1995). TD models: Modeling the world at a mixture of time scales. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 531–539.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

[Sutton et al., 1999] Sutton, R. S., Precup, D., and Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211.

[Sutton et al., 2005] Sutton, R. S., Rafols, E. J., and Koop, A. J. (2005). Temporal abstraction in temporal-difference networks. In *Advances in Neural Information Processing Systems 18*, pages 1313–1320.

[Sutton and Tanner, 2004] Sutton, R. S. and Tanner, B. (2004). Temporal-difference networks. In *Advances in Neural Information Processing Systems 17*, pages 1377–1384.

[Tanner and Sutton, 2005a] Tanner, B. and Sutton, R. S. (2005a). TD(λ) networks: Temporal-difference networks with eligibility traces. In *Proceedings of the Twenty-Second International Conference on Machine Learning (ICML)*, pages 889–896.

[Tanner and Sutton, 2005b] Tanner, B. and Sutton, R. S. (2005b). Temporal-difference networks with history. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 865–870.

[Tanner, 2005] Tanner, B. T. (2005). *Temporal-difference Networks*. PhD thesis, Department of Computer Science, University of Alberta, Edmonton, Alberta.

[Tesauro, 1995] Tesauro, G. (1995). Temporal difference learning and TD-gammon. *Communications of the ACM*, 38(3):58–68.

[Watkins, 1989] Watkins, C. (1989). *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, England.

[Wiewiora, 2005] Wiewiora, E. (2005). Learning predictive representations from a history. In *Proceedings of the Twenty-Second International Conference on Machine Learning (ICML)*, pages 964–971.

[Wingate and Singh, 2006a] Wingate, D. and Singh, S. (2006a). Kernel predictive linear gaussian models for nonlinear stochastic dynamical systems. In *Proceedings of the Twenty-Third International Conference on Machine Learning (ICML)*, pages 1017–1024.

[Wingate and Singh, 2006b] Wingate, D. and Singh, S. (2006b). Mixtures of predictive linear gaussian models for nonlinear stochastic dynamical systems. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 524–529.

[Wolfe et al., 2005] Wolfe, B., James, M. R., and Singh, S. (2005). Learning predictive state representations in dynamical systems without reset. In *Proceedings of the Twenty-Second International Conference on Machine Learning (ICML)*, pages 88–95.

[Wolfe and Singh, 2006] Wolfe, B. and Singh, S. (2006). Predictive state representations with options. In *Proceedings of the Twenty-Third International Conference on Machine Learning (ICML)*, pages 1025–1032.