MODEL-BASED REINFORCEMENT LEARNING USING INFORMATION-THEORETIC DESCRIPTORS

By

MATTHEW S. EMIGH

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2017

Dedicated to my parents

ACKNOWLEDGMENTS

First and foremost, I'd like to thank my advisor Jose Carlos Santos Carvalho Principe for taking me on and teaching me how to be researcher. My other committee members, John Shea, Scott McCullough, and Anand Rangarajan were all very helpful in my journey through grad school. I'd also like to thank Karl Gugel for introducing me to Dr. Principe and getting me a position in CNEL. Thanks to Tory Cobb for the help and advice working on ONR projects and bringing me to Panama City for the summer where I was able to grow a lot as a researcher.

I'd also like to thank all my CNEL labmates who helped me along the way. A special thanks to Austin Brockmeier, my original mentor, and Evan Kriminger who helped me through my worst moments in grad school. I can't thank these guys enough.

Thanks to Ryan Burt who read my papers and offered great suggestions. Thanks to Eder Santana for the enormous help and collaboration, and for teaching me deep learning. Thanks to Isaac Sledge for all the help he gave me toward the end. Thanks to Lin Li, Pingping Zhu, and Songlin Zhao for introducing and welcoming me to CNEL. Thanks to Carlos Loza for being good at soccer. Thanks to Subit Chakrabarti for being a cool guy. Thanks to Jihye Bae and Luis Gonzalo Sanchez Giraldo for answering my questions whenever I was stuck or needed help. Thanks to Goktug Cinar for the entertainment and moral support. Thanks to Kan Li and Gabriel Nallathambi for good advice. Thanks to Badong Chen for being so helpful whenever asked. Thanks to Bilal Fadlallah, Rakesh Chalasani, and Rosha Pokharel for the helpful discussions. Thanks to Jongmin Lee and In Jun Park for teaching me some Korean, and Miguel Teixeira for teaching me some Portuguese. Thanks to Arash Andalib for being so helpful and kind. Thanks to Zheng Cao, Ying Ma, Shujian Yu for being a good labmate.

I'd like to thank my friends from home: Clay Halbert, Gary Halbert, Jake Halbert, and Neil Dearwester for encouraging me to finish.

Last of all (but not least), I'd like to thank my Portuguese love, Catia Sofia Pinho da Silva for supporting me. Beijinhos.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

MODEL-BASED REINFORCEMENT LEARNING USING INFORMATION-THEORETIC DESCRIPTORS

By

Matthew S. Emigh

May 2017

Chair: Jose Principe
Major: Electrical and Computer Engineering

Reinforcement learning is a subfield of machine learning which attempts to quantify the value of each state or state/action combination in a space. For control problems, the goal is to select actions so as to maximize the expected cumulative rewards experienced by an agent visiting the states. This is a difficult problem for a number of reasons, and is especially so in continuous spaces. We propose a novel framework for exploring and exploiting continuous (infinite) spaces and mitigating the numerous difficulties associated with high dimensional and infinite spaces. We first discuss using metric learning to reduce the dimensionality of a state space and to use in combination with state approximation methods. We also discuss a novel method to compute transition probabilities in continuous state spaces. We use an information-theoretic learning approach to compute divergences between transition probability probability density functions to quantify uncertainty in the space. We then discuss using divergence and a novel concept called divergence-to-go in a model-based reinforcement learning framework to explore unknown regions of the continuous space. Finally, we conclude and suggest directions for future research.

CHAPTER 1
INTRODUCTION

Reinforcement learning (RL) is a well-established framework belonging to the field of machine learning, lying somewhere between the subfields of unsupervised and supervised learning. For supervised learning, given a set of samples $\{x_i, y_i\}$ belonging to an input space X and output space Y, the objective is to learn a function f that maps $X \to Y$. In contrast, given $\{x_i\}$, the goal of unsupervised learning is to learn a mapping $X \to Y$, using only the underlying structure of X.

RL algorithms are given a set of sample *sequences* $\{(x_1, r_1, x_2, r_2, \ldots)\}$, where r_i are rewards associated with $x_i \in X$. The main difference seen here between RL and supervised/unsupervised learning is that the sample data is ordered in time and therefore contains information in its time structure. In the RL framework, X is known as the state space, and $x \in X$ are known as states. Reinforcement learning algorithms can learn either passively or actively, i.e., can passively accept data samples, or can act to influence which samples are picked. For passive learning problems, the goal of these algorithms is to learn a function $V : X \to \mathcal{R}$ which gives the *expected cumulative sum of rewards* over all state sequences from each $x \in X$.

For control problems, the goal of RL is learn a policy $\pi : X \to A$, where A is a space of possible actions, which maximizes V for every $x \in X$. In this case, V can be maximized either explicitly by selecting actions using action conditioned value function $Q(x, a)$ [1] or implicitly by, e.g., optimizing a parameterized policy [2]. At any time in the process of selecting actions, the RL algorithm can either act to improve its policy by selecting samples it knows little about, or to exploit whatever knowledge it currently has. This is a fundamental problem in RL, and is known as the exploration/exploitation trade-off.

An underlying assumption for all RL algorithms is that there exists a joint probability distribution $P(x_1, x_2, \ldots)$ over all state sequences. This is clearly a difficult problem,

and to simplify things, the Markov assumption is made. That is, P can be decomposed by the chain rule as $P(x_1, x_2, \ldots, x_n) = P(x_n|x_{n-1}, ..x_1)P(x_{n-1}|x_{n-2}, \ldots, x_1) \cdots P(x_1) = P(x_n|x_{n-1})P(x_{n-1}|x_{n-2}) \cdots P(x_1)$. This "memoryless" property means that only the first order transition structure of state sequences need be used to compute the value function.

Reinforcement learning's roots lie in the work of Richard Bellman, performed in 1950s and 1960s, on dynamic programming. However, the father of Reinforcement Learning, Richard Sutton, undeniably kickstarted the field in 1988 with his seminal paper on temporal difference (TD) learning [3]. Since then, the field has exploded, with numerous applications in areas such as robot control [4], artificial intelligence [5], operations research [6], economics [7], and brain-machine interfaces (BMIs). In particular, the Computational Neuro-engineering Laboratory (CNEL) at the University of Florida introduced symbiotic BMIs [8] [9], which allow a neural decoder to learn motor neural states using reward signals generated by the Nucleus Accumbens. This is particularly important because disabled individuals (who would actually benefit from BMIs) are unable to produce the kinematic data samples required to train supervised learning algorithms.

One of the most celebrated success stories of RL is Tesauro's training of a neural network to become a world-class Backgammon player [10]. TD-gammon was able to learn and exploit strategies never before seen by human players. It turns out that the game of backgammon is particularly suited for being trained by RL for a number of reasons [11] [12] [13] [14]. However, similar, though not as exceptional, success stories have been achieved for the games of solitaire [15], chess [16], and checkers [17]. Furthermore Minh et al [18] were able to use deep learning and RL to train neural networks to successfully play atari games using the arcade learning environment [19]. Using RL techniques, Stone's [20] team has consistently trained agents to successfully compete and win in the annual Robocup robotic soccer competition. Ng et al [21]

were able to use RL to train a controller to sustain inverted flight with an autonomous helicopter. In the field of economics, Moody and Saffell [22] were able to train an RL agent to learn investment policies.

A recent successful trend in machine learning research applied to reinforcement learning is to use policy functions parameterized as deep neural networks. The representation power of multiple hidden layer architectures and advances in regularization techniques make them practical for learning complex control functions. A drawback for training such powerful networks is the amount of data they require at training phase. This problem is even more salient in RL because the agent slowly collects observation as it acts in the world. To compensate the ”strong representation power vs big data requirements” problem, deep reinforcement learning (DeepRL) literature relies either on experience replay [18] or asynchronous techniques where several agents with shared policies parameters learn while interacting with the world in parallel.

Pairing strong policy functions parameterized with deep neural networks and enough data for composing batches for neural network training led to the problem of weak exploration and overfitting. Once we use a powerful model for learning a control function, they are prone to overfitting, which is a common problem in supervised learning with deep neural networks. In RL the overfitting problem presents as networks that don't explore the state-action space thoroughly. A successful approach for encouraging exploration in deep RL is to use a regularization in the cost function that forces the policy function to have high entropy given an input [23]. This way, the network is punished if the action probability distribution is far from uniform. Other exploration schemes represent the agent/world interaction as a communication channel and try to maximize the amount of information transmitted from the world to the agent [24].

Despite its many successes, reinforcement learning is severely limited by its intrinsic shortcomings. Firstly, RL algorithms suffer greatly from the so-called “curse of dimensionality” [25], a term coined by Bellman. It describes the property that as the

dimensionality of the state space increases, its volume increases exponentially, causing the sample data to become exponentially sparse. RL algorithms typically require visiting each state $x \in X$ multiple times in order to learn the structure of state sequences. This quickly becomes intractable as the dimensionality of the state space, and therefore convergence time, increases. Unfortunately, many or most real-world domains can only be described by high-dimensional data. This motivates finding state representations of lower dimension that are relevant to reinforcement learning's goal of optimizing the value function.

Secondly, for the same reasons discussed above, many RL approaches are incapable of operating in continuous (infinite) state spaces. To alleviate this problem, many RL algorithms require that continuous input data first be discretized before being used to compute a policy and/or value function. This discretization must be performed by a human using knowledge of the input domain and is typically performed uniformly across the space. This is clearly not the most efficient nor precise way to perform discretization. For example, the state-reward dynamics may be relatively constant in some parts of the space, while they may change rapidly in other parts of the space. A better discretization would aggregate the similiar states while splitting up dissimilar ones.

Thirdly, in RL control problems, it is difficult to know how to choose actions to effectively learn the structure of the problem. It has been shown that by always selecting the greedy action, that is, selecting the action that maximizes the imperfectly known value function, leads to sub-optimal policies [5]. The most common approach to gather information about the problem is to randomly select actions. This approach is only optimal asymptotically, and can lead to long convergence times.

This dissertation attempts to mitigate the above difficulties by assembling a group of novel approaches into a common framework which achieve the following:

1. Find reduced-dimensionality state space representations that are still relevant to achieving the goal of maximizing the value function.

2. Create a model-based approach capable of learning the rewards and transition probabilities of a continuous-space RL problem

3. Intelligently direct exploration in a continuous state space in order to learn the structure of the problem

The rest of this dissertation is organized as follows: In Chapter 2, the background required for this work is discussed. This includes brief descriptions of model-free and model-based RL, information-theoretic learning, and the kernel least mean squares algorithm. Chapter 3 introduces kernel temporal difference learning and presents extensions using action kernels and experience replay. In Chapter 4, a novel method to learn new state representations using metric learning is presented. In Chapter 5, work in a model-based approach to directly learn and explore continuous state spaces is presented. In Chapter 6, we conclude and offer ideas for future work.

CHAPTER 2
BACKGROUND

2.1 Reinforcement Learning

2.1.1 Markov Decision Processes

Reinforcement learning (RL) [26, 27] is a natural paradigm for a machine to mimic the biological process of learning to complete objectives. It is applied to sequential decision problems, in which the outcome of an action may not be immediately apparent. With RL, credit can be assigned to the actions leading to an outcome, which provides feedback in the form of a reward. The feedback that occurs in response to actions is combined with previous experience. Thus, RL serves as a principled means of learning policies for decision making directly from experience.

Reinforcement learning is formulated for problems posed as Markov Decision Processes (MDP). An MDP is a model for a certain class of stochastic control problems. A (discounted) MDP can be represented mathematically as a 5-tuple $(\mathcal{X}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{X}$ is the set of all possible states (state space), $\mathcal{A}$ is the set of all possible actions (action space), $\mathcal{P}$ is the set of transition probabilities, $\mathcal{R}$ is the set of rewards, and γ is the discount factor. Since the available actions are typically state-dependent, we define $\mathcal{A}(x) = \{a \in \mathcal{A} | a \text{ is allowed in } x\}$. A state or action occurring at time t is denoted by X_t and A_t, where $X_t \in \mathcal{X}$ and $A_t \in \mathcal{A}$. When an agent takes an action a in a certain state x, this leads to a new state x'. This new state is a random variable, and a probability is assigned to the transition as follows: $P(x, a, x') = P(X_{t+1} = x' | X_t = x, A_t = a)$. Note that the next state probability is conditioned only on the current state and action. This is known as the Markov property, and is the primary assumption for modeling with an MDP. Rewards can be associated with states ($R(x')$) or state transitions ($R(x, x')$, $R(x, a, x')$). We write R_{t+1} to denote the reward received when the state transitions from X_t to X_{t+1} and $R(\cdot)$ to denote rewards associated with specific states or state transitions. Since

state transitions are random, rewards R_{t+1} are as well. Furthermore, rewards $R(x, a, x')$ may be random in their own right.

Since typically only a small fraction of possible state transitions receive a non-zero reward, credit must be assigned to states that indirectly led to the rewarded state. This is achieved by assigning a value to each state equal to the average sum of rewards received after visiting that state:

$$V^{\pi}(x) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | X_0 = x\right] \tag{2–1}$$

Of course, the value of each state is completely dependent on the policy $\pi : X \to A$ assigning an action (or action distribution for stochastic policies) to each state in X. The superscript π in V^{π} indicates that this value function is dependent on the policy chosen.

For control problems, it is helpful to define a value function that is dependent on the action. This is achieved by evaluating actions with the rewards that are received at current *and* future times, which is formalized in a criterion known as the Q-value. The value function is defined as the sum of expected future discounted rewards,

$$Q^{\pi}(x, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | X_0 = x, A_0 = a\right], 0 \leq \gamma < 1. \tag{2–2}$$

starting in state x and performing action a. γ is a discounting factor, which decays rewards received further in the future. This is the discounted value function; there are also value functions based on time-averaged rewards and on finite time-horizon sums of rewards. Note that the value function V can be obtained from the Q function by:

$$V^{\pi}(x) = Q^{\pi}(x, \pi(x)) \tag{2–3}$$

The expectation in (2–2) is taken over the transition probabilities of the MDP and under the policy, π, which is a mapping from states X to actions A. The policy may be stochastic or deterministic. If the policy is stochastic, then it is a mapping from the states X to a distribution over A. The MDP is considered to be solved when the policy

maximizes the expected future sum of rewards from any state $X \in \mathcal{X}$. We call $\pi^\star$ the optimal policy for the MDP. Every MDP has at least one optimal policy $\pi^\star$ [27].

It is intractable to calculate the value functions of MDPs directly because (2–2) requires expectations over all future times. The tools of dynamic programming (DP) [25] allow the value functions to be computed efficiently by using recursion. If (2–2) is rewritten as

$$Q^\pi(x, a) = R_1 + E\left[\sum_{t=1}^{\infty} \gamma^t R_{t+1} | X_0 = x, A_0 = a\right], \tag{2–4}$$

it is clear that the summation now represents the value function at the next state $\gamma Q^\pi(X_1, A_1)$. We can thus rewrite (2–2) as

$$Q^\pi(x, a) = E[R_{t+1} + \gamma Q^\pi(X_{t+1}, A_{t+1}) | X_t = x, A_t = a] \tag{2–5}$$

$$= \sum_{x' \in \mathcal{X}} P(x, a, x') \left(R_{t+1}(x, a, x') + \gamma Q^\pi(x', \pi(x'))\right) \tag{2–6}$$

where R_{t+1} is dependent on x_t and a_t. In (2–5) the expectation is now over a transition from a single state. DP computes the value functions using iterative methods such as value iteration and policy iteration where the optimal policies can be found by choosing actions which maximize the value function at each state:

$$Q^\pi_{k+1}(x, a) \leftarrow \sum_{x' \in \mathcal{X}} P(x, a, x') \left(r_{t+1}(x, a, x') + \gamma Q^\pi_k(x', \pi(x'))\right), \quad \forall x \in \mathcal{X} \tag{2–7}$$

$$\pi_{m+1}(x) \leftarrow \operatorname*{argmax}_a Q^{\pi_m}(x, a). \tag{2–8}$$

Equation 2–7 gives the recursive value iteration update, while Equation 2–8 gives the policy iteration update. The policy update can be run every value iteration update or some multiple of it.

However, even with the recursion of DP, the state transition probabilities as well as the rewards for each transition must be known. For real problems with very large state spaces, this is an unreasonable assumption.

2.1.2 Model-Free Reinforcement Learning

A subset of reinforcement learning algorithms learn a policy $\pi : \mathcal{X} \to \mathcal{A}$ without ever learning P and R. These algorithms include Q-learning and SARSA.

Q-learning [1] is an RL algorithm that learns the optimal Q-function, $Q^\star(x, a)$, for each state-action pair. The optimal policy $\pi^\star$ can be written in terms of the optimal Q-function as $\pi^\star(x) = \arg\max_a Q^\star(x, a)$. Q-learning solves $E[R_{t+1} + \gamma \max_a Q(X_{t+1}, a)]$ using stochastic approximation [28], thus removing the need to calculate expectations and learning Q-functions directly from observed $(X_t, A_t, R_{t+1}, X_{t+1})$. The classical Q-learning algorithm is defined by the following update:

$$\begin{aligned} Q(x_t, a_t) &\leftarrow Q(x_t, a_t) + \alpha\delta \\ \delta &= r_{t+1} + \gamma \max_{a'} Q(x_{t+1}, a') - Q(x_t, a_t) \end{aligned} \tag{2–9}$$

where α is a small learning rate, and $Q(\cdot, \cdot)$ is the current Q-function estimate.

Suppose $x, y \in \mathcal{X}$ and $a \in \mathcal{A}$, the action a is taken in x, and the state transition $x \to y$ is observed. Furthermore, suppose reward r_{xy} is received. Then the term $r_{xy} + \gamma \max_{a'} Q(y, a')$ in (2–10) is a bootstrapped observation of $Q^\star(x, a)$. The sample averages of these observations will converge to $Q^\star(x, a)$ for all x, a if each possible state transition is observed an infinite number of times. Q-learning is an on-line algorithm which uses these observations to stochastically update its estimate of $Q^\star(x, a)$.

SARSA is another prominent model-free RL algorithm. It is similar to Q-learning, but it learns according to

$$\begin{aligned} Q(x_t, a_t) &\leftarrow Q(x_t, a_t) + \alpha\delta \\ \delta &= r_{t+1} + \gamma Q(x_{t+1}, a_{t+1}) - Q(x_t, a_t) \end{aligned} \tag{2–10}$$

The Q-value estimate for SARSA is for the current policy. In contrast, the Q-value estimate for Q-learning is for the optimal policy. Under certain conditions and learning policies [29], the SARSA Q-function will converge to the optimal Q-function with probability 1.

The update in (2–10) is known as tabular Q-learning, since a Q-value for each state-action pair is stored in a table and updated each time that pair is visited by the agent. For problems with large or continuous state-action spaces there are so many states (state-actions) that there is not enough experience to provide good estimates of the Q-functions in the table. In this case, a function approximator which maps state-actions to Q-functions is learned from the data. However, theoretical guarantees for Q-learning in the general case, in which no assumptions are made about the sampling distribution of the observed state-action pairs, are limited. The majority of work in this area has focused on the linear case, where $Q_\theta(x, a) = \theta^T \phi(x, a)$, and θ is a d dimensional parameter vector and $\phi(x, a)$ is the feature vector representation of the state-action pair [26]. For practical problems, nonlinear function approximators such as neural networks can be employed to learn the value function, however these methods can be quite unreliable.

2.1.3 Model-Based Reinforcement Learning

Q-learning and SARSA, which are known as model-free RL algorithms, learn a policy without ever learning P and R. They learn exclusively by learning a Q-function and choose a policy by selecting actions that maximize said Q-function. In contrast, there exists another reinforcement learning framework known as model-based RL. Model-based RL methods model the underlying MDP by learning or approximating P and R. Once a model of the MDP is known, a policy π can be formulated offline using methods such as value iteration and policy iteration. Naturally, model-based methods require more memory and computational resources, but have the ability to learn a policy much more quickly in terms of number of actions that must be taken and number of

Table 2-1. Q-learning with ϵ-greedy exploration

1) Initialize parameters γ, α, ϵ
2) Repeat:
3) $\epsilon_0 \leftarrow$ random number between 0 and 1
4) $x \leftarrow$ current state
5) if $\epsilon_0 > \epsilon$ then:
6) $a \leftarrow \text{argmax}_{a'} Q(x, a')$
7) else:
8) $a \leftarrow$ random action
9) Take action a, observe state transition
10) $x' \leftarrow$ new state
11) $r \leftarrow$ reward associated with state transition
12) $\delta \leftarrow r + \gamma \max_{a'} Q(x', a') - Q(x, a)$
13) $Q(x, a) \leftarrow Q(x, a) + \alpha \cdot \delta$

times states must be visited in X [30]. For example, in temporal-difference based model-free learning, values must be propagated back through the space as state transitions are experienced.

For small finite-state MDPs, the simplest modeling approach is to approximate the rewards and transition probabilities using a simple maximum likelihood estimate. For example, the reward associated with state x' can be computed as $\frac{1}{N_{x'}} \sum_i R_i(x')$, where $N_{x'}$ is the number of times state x' has been visited and $R_i(x')$ is the reward received the i^{th} visit. Furthermore, $P(x, a, x')$ can be computed as $\frac{N_{x,a,x'}}{N_{x,a}}$, where $N_{x,a}$ is the number of times action a has been taken in state x and $N_{x,a,x'}$ is the number of times x' has been reached after taken action a in state x. Clearly, this approach becomes impractical as the number of states and actions increase.

For larger state and action spaces, models which are able to generalize are required. Schaal and Atkeson [31] used locally weighted regression to train a "deterministic" inverse $\hat{a}_k = \hat{f}^{-1}(x_k, x_{k+1})$ and forward transition model $\hat{x}_{k+1} = \hat{f}(x_k, \hat{a}_k)$. A linear-quadratic controller was then computed for each local linear model. Using this

framework, they were able to teach a robot to juggle "devil sticks." Hester and Stone [32] use decision trees to generalize a transition and reward model. However, this framework requires discrete, factored [33] MDPs . Deisenroth [34] used Gaussian process regression to learn a model, but the computational requirements are extremely high. Jong and Stone [35] created a transition and reward model similar to the work in this paper using averaging kernels. However, their work was not from an information-theoretic point of view, and is unable to exploit the catalog of algorithms and techniques that have been developed within the information-theoretic learning [36] framework.

2.1.4 Model Directed Exploration

Another advantage of model-based reinforcement learning is that it can be used to improve sample efficiency. That is, while online, the model can help direct exploration of the space in such a way as to improve itself. This is especially important in the case of infinite state spaces since it is impossible to visit every state and difficult to even revisit previously experienced states. In the case of the simple empirical modeling approach described above, many exploration mechanisms, such as E3 [37] and Rmax [38], have been proposed. Unfortunately, such approaches require visiting every state exhaustively.

2.2 Information-Theoretic Learning

Information-theoretic learning (ITL) is a framework, pioneered by Principe [36], that computes information-theoretic quantities and uses them to train learning machines. This framework computes and optimizes information-theoretic descriptors as opposed to the second-order statistics conventionally used in statistical learning. The foundation of ITL is built upon Renyi's (differential) α-order entropy, which is given by

$$H_\alpha(X) = \frac{1}{1-\alpha} \log \int p^\alpha(x)\, dx, \qquad \alpha \geq 0, \alpha \neq 1 \tag{2–11}$$

where p is a probability density function (pdf) belonging to L^α. Renyi's entropy is used, as opposed to Shannon's $H(X) = -\int p(x) \log p(x)\, dx$, because it can easily be estimated nonparametrically from pairwise samples. For the case of $\alpha = 2$, Equation

(2–11) is known as Renyi's quadratic entropy which is given by $H_2 = -\log \int p^2(x)dx$. Given samples x_j drawn from p, the Gaussian kernel $\left(G_\sigma = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{\|x-y\|^2}{2\sigma^2}\right)\right)$ Parzen window estimate of a probability density function (pdf) is given by

$$\hat{p}_X(x) = \frac{1}{N\sigma} \sum_{i=1}^{N} G_\sigma \left(\frac{x - x_i}{\sigma}\right). \tag{2–12}$$

Renyi's quadratic entropy can be computed by plugging (2–12) into (2–11):

$$\hat{H}_2(X) = -\log \int_{-\infty}^{+\infty} \left(\frac{1}{N} \sum_{i=1}^{N} G_\sigma(x - x_i)\right)^2 dx \tag{2–13}$$

$$= -\log \left(\frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} G_{\sigma\sqrt{2}}(x_j - x_i)\right). \tag{2–14}$$

The argument of the log in Equation 2–13 is called the *information potential* in analogy with potential fields from physics and is denoted by $\hat{V}_\sigma(X)$. The information potential can often be used in place of Renyi entropy since the logarithm is a monotonic function. For example, computing the gradient of the argument with respect to some parameter vector that induces it is simpler than computing it with respect to 2–13.

Other information-theoretic quantities can be computed as well, such as divergences. In particular, Principe proposed a divergence he named "Cauchy-Schwarz divergence" since it was derived from the Cauchy-Schwarz inequality. The CS-divergence between two pdfs p_1 and p_2 is defined as:

$$D_{CS}(p_1, p_2) = -\log \frac{\left(\int p_1(x)p_2(x)\,dx\right)^2}{\int p_1^2(x)\,dx \int p_2^2(x)\,dx}. \tag{2–15}$$

CS divergence is symmetric and satisfies $0 \leq D_{CS} < \infty$ where the zero is only obtained if $p_1 = p_2$.

Another important ITL divergence is the Euclidean divergence which is defined as the Euclidean distance between pdfs. The Euclidean divergence between pdfs p_1 and p_2

is defined as

$$D_{ED}(p_1\|p_2) = \int \left((p_1(x) - p_2(x)\right)^2 dx \tag{2–16}$$

$$= \int p_1^2(x)dx + \int p_2^2(x)dx - 2\int p_1(x)p_2(x)dx \tag{2–17}$$

Euclidean divergence has been shown to be a lower bound for Kullback-Leibler divergence [39].

Using the Parzen estimator described above 2–12 and assuming $\{x_1(i)\}_{i=1}^N$ and $\{x_2(i)\}_{i=1}^M$ are drawn from p_1 and p_2 respectively, Cauchy-Schwarz and Euclidean divergences can be computed using the equations:

$$\hat{V}_f = \sum_{i=1}^{N}\sum_{j=1}^{N} G_{\sigma\sqrt{2}}(x_1(i), x_1(j)) \tag{2–18}$$

$$\hat{V}_g = \sum_{i=1}^{M}\sum_{j=1}^{M} G_{\sigma\sqrt{2}}(x_2(i), x_2(j)) \tag{2–19}$$

$$\hat{V}_c = \sum_{i=1}^{N}\sum_{j=1}^{M} G_{\sigma\sqrt{2}}(x_1(i), x_2(j)) \tag{2–20}$$

and

$$\hat{D}_{cs} = \log \frac{\hat{V}_f \hat{V}_g}{\hat{V}_c^2} \tag{2–21}$$

$$\hat{D}_{ED} = \hat{V}_f + \hat{V}_g - 2\hat{V}_c. \tag{2–22}$$

Mutual information can also be computed using both types of divergences. For random variables $X_1, X_2, \ldots, X_n$ drawn from the joint distribution pdf $p(X_1, X_2, \ldots, X_n)$, the Cauchy-Schwarz mutual information can be computed as:

$$I_{CS}(X_1, X_2, \ldots, X_n) = D_{CS}\left(p(x_1, x_2, \ldots, x_n), \prod_{i=1}^{n} p_{X_i}(x_i)\right) \tag{2–23}$$

where p_{X_i} is the marginal probability density function for X_i. Similarly,

$$I_{ED}(X_1, X_2, \ldots, X_n) = D_{ED}\left(p(x_1, x_2, \ldots, x_n), \prod_{i=1}^{n} p_{X_i}(x_i)\right). \tag{2–24}$$

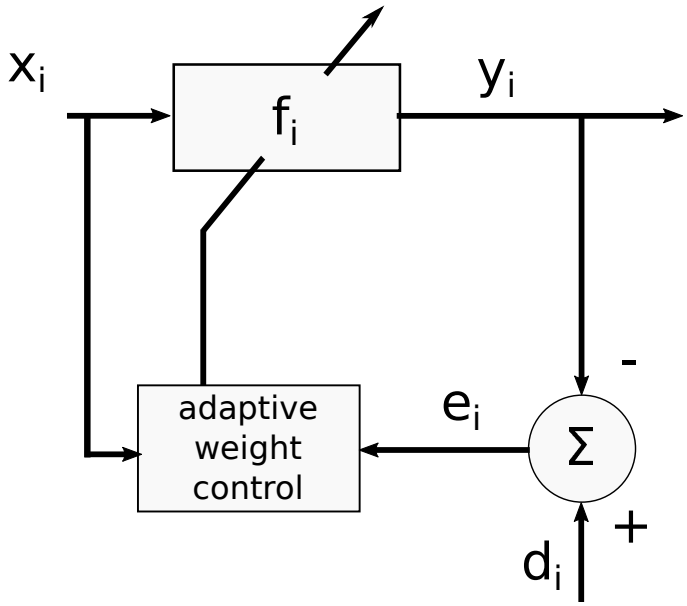

Figure 2-1. Non-linear adaptive filter block diagram. The variables x and y respectively represent the input and output to the non-linear filter f. The variable d represents the desired value the filter f is trying to learn. The desired d and filter output y are subtracted to compute the error e, which is used by the adaptive weight control. The adaptive weight control block adjusts the non-linear function f. The subscript i for each variable represents the value at time (or iteration) i.

2.3 Function Approximation

Kernel adaptive filters (KAFs) [40] are nonlinear adaptive filters that use kernel methods to learn a mapping f. In general, KAFs (Figure 2-1) sequentially update their mapping according to $f_i = f_{i-1} + G(i)e(i)$, where $e(i)$ is the error in the mapping computed at iteration i and $G(i)$ is some gain function. The Kernel Least Mean Squares (KLMS) algorithm is one type of KAF and is furthermore a convex universal learning machine (CULM). That is, it is a universal approximator trained using convex optimization.

KLMS is based on the classical Least Mean Squares (LMS) algorithm [41] which iteratively updates its weights by minimizing the instantaneous squared error $J_i = \frac{1}{2}e_i^2 = \frac{1}{2}(d_i - w_{i-1}^T x_i)^2$. The gradient of this cost function with respect to the weights is

$$\nabla_{w_{i-1}} J(i) = -e_i x_i \tag{2–25}$$

By the method of gradient descent, LMS updates its weights at each iteration according to

$$w_i = w_{i-1} + \eta e_i x_i, \tag{2–26}$$

where η is some constant, usually referred to as step size or learning rate.

The LMS update can be rewritten in terms of inner products. Assuming $w_0 = 0$, and expanding the update equation:

$$\begin{aligned} w_1 &= \eta e_1 x_1 \\ w_2 &= \eta e_1 x_1 + \eta e_2 x_2 \\ &\dots \\ w_n &= \eta \sum_i e_i x_i. \end{aligned} \tag{2–27}$$

Then the output can be written as

$$\begin{aligned} y &= x^T \left(\eta \sum_i e_i x_i \right) \\ &= \eta \sum_i e_i \left\langle x_i, x \right\rangle. \end{aligned} \tag{2–28}$$

The Kernel Least Mean Squares algorithm is simply the LMS algorithm performed on data that has been mapped into a (possibly countably infinite dimensional) Reproducing Kernel Hilbert Space (RKHS).

Let X be an arbitrary set. An RKHS can be defined as a Hilbert space of (real-valued) functions $f : X \to \mathbb{R}$ for which all evaluation functionals $L_x : f \mapsto f(x)$ are bounded and linear. Since all evaluation functionals are bounded and linear, the Riesz representation theorem guarantees that $\forall x \in X$, there exists a unique $K_x \in H$ such that $\forall f \in H$, $f(x) = L_x(f) = \langle f, K_x \rangle$. That is, every evaluation functional can be represented as an inner product in the RKHS. This is known as the reproducing property from which

the RKHS takes its name. Furthermore,

$$K_y(x) = L_x(K_y) = \langle K_y, K_x \rangle = \langle K_x, K_y \rangle = L_y(K_x) = K_x(y). \tag{2–29}$$

We define the function $K(x, y) = K_y(x) = K_x(y)$, which we refer to as the reproducing kernel. Clearly $K(x, y)$ is symmetric. It is also positive definite, i.e, for any $x_1 \dots x_n \in X$, $c_1, \dots, c_n \in \mathbb{R}$, then $\sum_{i,j} c_i c_j K(x_i, x_j) \geq 0$ as can be seen by

$$\sum_i \sum_j c_i c_j K(x_i, x_j) = \sum_i \sum_j c_i c_j \left\langle K_{x_i}, K_{x_j} \right\rangle = \left\langle \sum_i c_i K_{x_i}, \sum_j c_j K_{x_j} \right\rangle = \left\| \sum_i c_i K_{x_i} \right\|_H^2 \geq 0. \tag{2–30}$$

An RKHS can be alternatively defined in terms of its reproducing kernel. That is, an RKHS is a Hilbert space of functions $f : X \to \mathbb{R}$ (X non-empty) with a reproducing kernel K whose span is dense in H.

It can be challenging to build an RKHS from scratch, as it is difficult to know beforehand which Hilbert spaces of functions satisfy the necessary properties. Fortunately, the Moore-Aronszajn theorem states that each symmetric positive definite kernel $k : X \times X \to \mathbb{R}$ defines a unique RKHS where $K_x = k(x, \cdot)$. That is, in order to gain access to all the useful properties of an RKHS, one needs only to select a symmetric positive definite kernel.

Suppose X is a measurable space with a finite measure μ. Mercer's theorem then states that any symmetric real-valued positive definite kernel $k \in L_\infty(X \times X)$ can be decomposed as:

$$k(x, x') = \sum_i^N \lambda_i \phi_i(x) \phi_i(x'), \tag{2–31}$$

for almost all $x \in X$. When $N = \infty$, convergence is absolute and uniform. The variables λ_i and ϕ_i are the eigenvalues and corresponding eigenfunctions of the linear operator $T_k : L_2(X) \to L_2(X)$, where $(T_k f)(x) = \int k(x, z) f(z) d\mu(z)$ and $f \in L_2(X)$. This allows

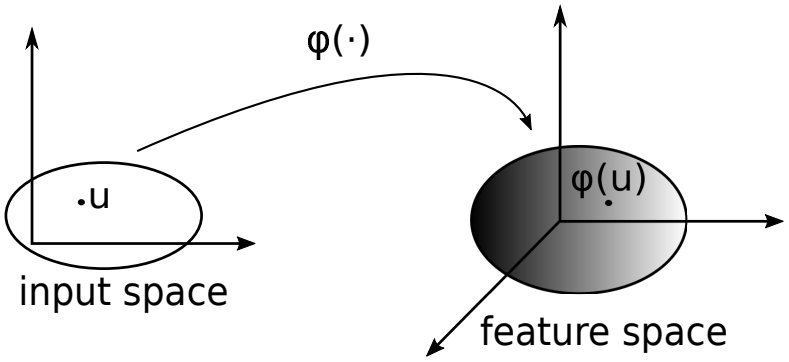

Figure 2-2. Non-linear feature mapping. The input u is mapped non-linearly into the feature space by way of the function $\varphi(\cdot)$. The feature space is constructed via Mercer's Theorem from the RKHS.

the mapping φ to be constructed

$$\varphi : X \to \mathbb{F} = \ell^2 \tag{2–32}$$

$$\varphi(x) = \left[\sqrt{\lambda_1}\phi_1(x), \sqrt{\lambda_2}\phi_2(x), \ldots\right], \tag{2–33}$$

which we treat as a mapping from the input space to some (possibly infinite dimensional) feature space $\mathbb{F}$. That is, $\varphi(x)$ is a (possibly infinite dimensional) feature vector. Note that,

$$\varphi(x)^T\varphi(y) = \langle\varphi(x), \varphi(y)\rangle = \sum_i \lambda_i\phi_i(x)\phi_i(y) = k(x, y). \tag{2–34}$$

Also,

$$\varphi(x) = k(x, \cdot) \tag{2–35}$$

The KLMS update can now be computed by performing the LMS update on data that has been transformed into the feature space $\varphi : X \to \mathbb{F}$. Following the progression in 2–27, the weights in the feature space can be computed as:

$$\omega_i = \eta \sum_i e_i\varphi(x_i). \tag{2–36}$$

For infinite-dimensional feature spaces, e.g., the feature space induced by the Gaussian kernel, the infinitely long vector ω_i cannot be directly accessed or manipulated.

Nevertheless the filter output can be computed via the kernel trick:

$$\omega_i^T \varphi(x') = \left[\eta \sum_j^i e_j \varphi(x_j)^T \right] \varphi(x') \tag{2–37}$$

$$= \eta \sum_j^i e_j \left[\varphi(x_j)^T \varphi(x') \right] \tag{2–38}$$

$$= \eta \sum_j^i e_j k(x_j, x'). \tag{2–39}$$

Alternatively, the algorithm can be computed in the RKHS by minimizing the squared error according to:

$$J_i = \frac{1}{2} \left(d_i - \omega_{i-1}^T \varphi(x_i) \right)^2. \tag{2–40}$$

Taking the frechet derivative with respect to ω_{i-1},

$$\nabla J_i = -e_i \varphi(x_i). \tag{2–41}$$

Once again, following the method of gradient descent, we have:

$$\omega_i = \omega_{i-1} + \eta e_i \varphi(x_i). \tag{2–42}$$

Then, by following the same logic as (2–27)

$$\omega_n = \eta \sum_i^n e_i \varphi(x_i). \tag{2–43}$$

Finally, notice that

$$\|\varphi(x)\|^2 = \varphi(x)^T \varphi(x) = k(x, x), \tag{2–44}$$

and that for the Gaussian kernel $k(x, x) = 1$. Thus the normalized KLMS weight update can be given as

$$\omega_i = \omega_{i-1} + \frac{\eta}{k(x_i, x_i)} e_i \varphi(x_i), \tag{2–45}$$

which for the Gaussian kernel, reduces to regular update (2–42). This shows that using KLMS with a Gaussian kernel is automatically normalized. The KLMS algorithm can

thus be summarized as:

$$f_{i-1} = \eta \sum_{j=1}^{i-1} e_j k(x_j, \cdot) \tag{2–46}$$

$$f_{i-1}(x_i) = \eta \sum_{j=1}^{i-1} e_j k(x_j, x_i) \tag{2–47}$$

$$e_i = d_i - f_{i-1}(x_i) \tag{2–48}$$

$$f_i = f_{i-1} + \eta e_i k(x_i, \cdot) \tag{2–49}$$

where x_i are the input samples, d_i are samples of the desired or "target" function, and η is a fixed learning rate parameter. The kernel k is typically selected to be the Gaussian kernel, although other kernels capable of universal approximation are available. An advantage of KLMS is that it does not require explicit regularization as shown by Liu [42].

Unlike the classical LMS algorithm, KLMS requires storing an indefinitely growing dictionary of input and error values to be able to compute the next input. To curb the growth of this dictionary, a variety of quantization methods have been proposed. One such method has been named the quantized kernel least mean squares (QKLMS) algorithm. QKLMS modifies Equation (2–49) according to

$$f_i = f_{i-1} + \eta e(i) k(Q(u(i)), \cdot), \tag{2–50}$$

where Q i a quantization operator.

More specifically, QKLMS stores a dictionary, or codebook, matching inputs to error values, which according to (2–50) is all that is required to compute $f_i(x)$. We denote the codebook at iteration i as C_i. Each iteration, the distance between the new input and the codebook is computed:

$$d(x_i, C_{i-1}) = \min_{1 \le j \le \text{size}(C_{i-1})} \|x_i - C_{i-1}\| \tag{2–51}$$

If $d(x_i, C_{i-1})$ is less than some predetermined value ϵ_x, the error associated with the input corresponding to $\text{argmin}_{1 \leq j \leq \text{size}(C_{i-1})} \|x_i - C_{i-1}\|$ is updated according to Equation (2–50). That is, for stored error value j at iteration i,

$$E_i(j) = E_{i-1}(j) + \eta e_i k. \tag{2–52}$$

Otherwise, the input x_i and its associated error is entered into the codebook. Badong et al [43] showed empirically that, given an appropriate value for ϵ_x, QKLMS can achieve performance comparable or near identical to KLMS, while achieving codebooks an order of magnitude smaller.

CHAPTER 3
KERNEL TEMPORAL DIFFERENCES

The Kernel Temporal Difference algorithm, introduced by Bae et al. [44] updates the value function estimation using the KLMS update and temporal difference error,

$$e_{TD_i} = r_{i+1} + \gamma V(x_{i+1}) - V(x_i), \tag{3–1}$$

where $V(\cdot)$ is computed in the RKHS according to

$$V(x) = \omega^T \varphi(x) \tag{3–2}$$

That is, for each state in the state space $x \in X$, it first maps x to $\varphi(x)$ and then computes V(x) according to

$$V(x) = \eta \sum_j e_{TD_j} \varphi(x_j)^T \varphi(x), \tag{3–3}$$

where e_{TD} is temporal difference error.

For KTD(λ), the last two KLMS equations can be rewritten as

$$e_i = r_i + f_{i-1}(x_i) - f_{i-1}(x_{i-1}) \tag{3–4}$$

$$f_i = f_{i-1} + \eta e_i \sum_{j=1}^{i} \lambda^{i-j} k(x_j, \cdot) \tag{3–5}$$

where r_i is the immediate reward discussed above in Section 2.1, f is the value function estimate, and lambda is the eligibility trace parameter.

Q-learning requires learning a Q-function which has both state and action inputs. For the algorithm Q-ktd(λ), Bae essentially keeps separate KLMS models for each action:

$$Q(x_n, a = i) = Q_i(x_n) = \eta \sum_j e_j I_j(k) k(x_n, x_j), \tag{3–6}$$

where $I_j(k)$ is an indicator vector with k entries, and k is the number of discrete actions. The TD error was computed as:

$$e_i = r_i + \gamma \max_a Q_a(x_{n+1}) - Q_i(x_n) \tag{3–7}$$

Bae showed that KTD(λ), combined with ϵ-greedy exploration could be successfully applied to neural decoding of a reaching task on a monkey. Furthermore she showed that it could be used to solve typical RL test bed problems such as cart-pole and mountain-car.

3.1 KTD Algorithm with Action Kernel

Alternatively, we introduce a modified Q-ktd update that does not require keeping separate models for each action, and is applicable to environments with continuous actions. As opposed to the indicator vector used in Equation (3–6), we introduce the action kernel. The action kernel $k_a(\cdot, \cdot)$ can be any valid symmetric positive definite kernel. The kernel should be chosen to relate actions in a sensible way. A reasonable choice is often the Gaussian kernel which relates actions according to $G_{\sigma_a}(a, a') = \exp\left\{-\frac{\|a-a'\|^2}{2\sigma^2}\right\}$.

According to the Moore-Aronszajn theorem, the action kernel induces an RKHS H_a such that $k_a(a, \cdot) = \varphi_a(a)$. Furthermore, the kernel defined by the product $k((x, a)(x', a')) = k_x(x, x')k_a(a, a')$ is itself positive definite and induces an RKHS isometrically isomorphic to the tensor product of spaces $H_x \otimes H_a$, and $k((x, a)(\cdot, \cdot)) = \varphi_x(x) \otimes \varphi_a(a)$.

We perform the Q-update using this new kernel function:

$$Q(x_n, a) = \eta \sum_j^n e_j k((x, a)(x_j, a_j)), \tag{3–8}$$

where,

$$e_i = r_i + \gamma \max_a Q(x_{n+1}, a) - Q_i(x_n, a_n) \tag{3–9}$$

For discrete action spaces and the Gaussian kernel G_σ, this reformulation of Q-ktd can be made functionally identical to Bae's formulation by setting the kernel size σ to a very small value. In this case

$$G_{\sigma=small}(a, a') \approx \begin{cases} 0 & \text{if } a \neq a', \\ 1 & \text{if } a = a' \end{cases}, \tag{3–10}$$

which is equivalent to the indicator function.

3.2 Experience Replay

Experience replay [45] is a method to update an online reinforcement learning agent's value function by virtually revisiting state transitions stored in memory. It has been used recently, along with deep neural networks, with great success to learn to play Atari 2600 games [18]. Experience replay breaks temporal correlations by updating the model using samples (potentially) far in the past, and allows for the possibility of rare events being experienced more than once [46].

Experience replay is a simple concept. A sliding window of state transitions are stored in memory, i.e., state x_t, action a_t, next state, x_{t+1}, and reward r_{t+1}. At each iteration, an index vector I of length B is randomly selected (without replacement) from the $\{0, 1, \ldots, D\}$, where B is the batch size, and D is dictionary length. The value function is batchwise updated according to:

$$e_q^I \leftarrow e_q^I + \frac{\alpha}{B}\left(r_{t+1}^I + \gamma \max_a Q(x_{t+1}^I, a) - Q(x_t^I, a_t^I)\right), \tag{3–11}$$

Where e_q^I is the error vector corresponding to the k-qrl codebook positions of the index vector I, x_t^I, a_t^I, x_{t+1}^I, and r_{t+1}^I, are the corresponding states, actions, next states, and rewards respectively. We divide the learning rate by the batch size because large batch updates can cause the Q-function estimate to diverge. To see this, consider that with each experience replay update the weights in the RKHS are updated using gradient

Table 3-1. Q-ktd with experience replay

1) Initialize parameters N, γ, α, B, M
2) for i = 1 to N:
4) $x_i \leftarrow$ current state
5) Select action:
6) if explore:
7) $a_i \leftarrow$ exploratory action
8) if exploit:
9) $a_i \leftarrow \text{argmax}_{a'}\, Q(x, a')$
10) Take action a_i, observe state transition
11) $x_{i+1} \leftarrow$ new state
12) $\text{done}_{i+1} \leftarrow$ terminal state
13) $r \leftarrow$ reward associated with state transition
14) $H \leftarrow \{H, x_i, a_i, x_{i+1}, r_{i+1}, \text{done}_{i+1}\}$
15) if sequential update:
16) $\delta \leftarrow r_{i+1} + (1 - \text{done}_{i+1})\gamma \max_{a'} Q(x_{i+1}, a') - Q(x_i, a_i)$
17) $Q(x_i, a_i) \leftarrow Q(x_i, a_i) + \alpha \cdot \delta$
18) if i mod M == 0:
19) $I \leftarrow$ random index vector of size B
20) $\delta^I \leftarrow \frac{1}{B}\left(r^I_{t+1} + (1 - \text{done}^I)\gamma \max_a Q(x^I_{t+1}, a) - Q(x^I_t, a^I_t)\right)$
21) $Q(x^I, a^I) \leftarrow Q^I(x^I, a^I) + \alpha \cdot \delta^I$

descent according to

$$\omega_i = \omega_{i-1} + \alpha \sum_{n \in I} e_n \varphi(x_n). \tag{3–12}$$

Thus, dividing by the batch size is equivalent to averaging many gradient updates as opposed to simply adding them together. For large batch sizes the weight updates can grow quite large and overwhelm the small learning rate. Thus, it is advisable to normalize this gradient proportional to the number of experiences updated.

Experience replay is particularly well suited to q-krl because the algorithm already sequentially stores states in a dictionary. That is, in order to update the value function,

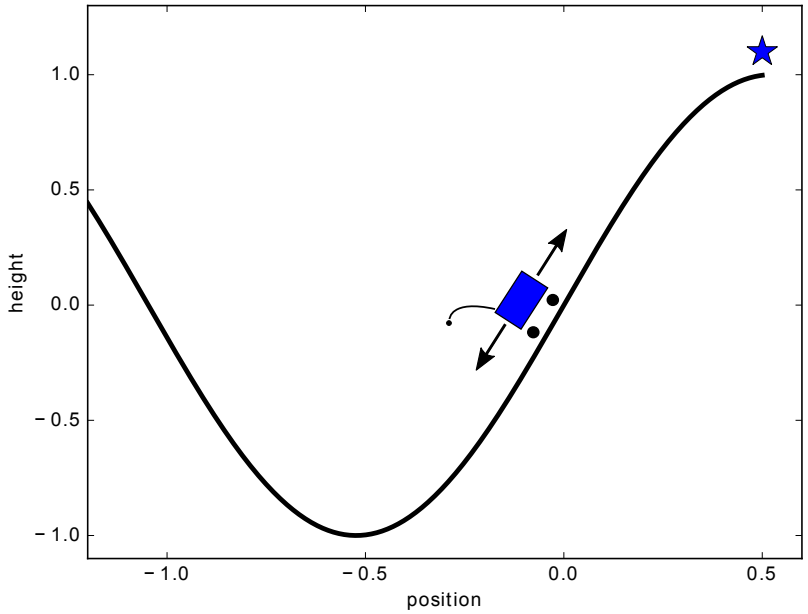

Figure 3-1. Mountain car problem. The agent (blue car) must traverse the hill to reach the goal position (blue star) starting at rest at the bottom of the hill. The mountain car has a motor that can move left or right, but it does not have enough power to drive directly to the top of the hill. Thus, it must first climb the left hill and then use the combination of its own power and gravity to climb the hill to the right.

elements of the codebook can be randomly selected and the corresponding error updated very quickly.

If quantized q-krl is used, the experience replay update becomes slightly more complicated. In this case, not every state is stored in the codebook, so an auxiliary index vector l_A is maintained that stores, for every state transition, the corresponding position in the codebook. Then the value function is updated according to:

$$e_q^{l_A} \leftarrow e_q^{l_A} + \frac{\alpha}{B}\left(r_{t+1}^{l} + \gamma \max_a Q(x_{t+1}^{l_A}, a) - Q(x_t^{l_A}, a_t^{l_A})\right), \tag{3–13}$$

3.2.1 Mountain Car

The mountain car problem [5] is a commonly used RL test bed in which the agent attempts to drive an underpowered car up a 2-dimensional hill. The problem is depicted

in Fig 3-1. The car begins at the bottom of the hill (position -0.5) and the agent attempts to drive it to the goal position at the top right. However, the car lacks sufficient power to drive directly to the goal, so it first must climb the left hill and allow the combination of gravity and the car's power to drive it up the right hill.

This problem can be fully described by only two variables: the position p and the velocity v of the mountain car. Thus, the state space $\mathcal{X}$ consists of these two variables. Depending on the set up, the action set consists of $\mathcal{A}_{disc} = \{-1, 0, +1\}$ or $\mathcal{A}_{cont} = [-1, 1]$. Each time step, the agent selects an action from $\mathcal{A}$ based on the 2-dimensional state space consisting of the cart's position and velocity. Negative actions drive the cart to the left, while positive actions drive it to the right. Typically, each time step is assigned a negative reward of -1 except for time steps associated with the goal state, which is assigned a reward of +1. The dynamics of the car are governed by the following equations:

$$v(n+1) = v(n) + 0.001a(n) - 0.0025\cos(3p(n)) \tag{3–14}$$

$$p(n+1) = p(n) + v(n+1). \tag{3–15}$$

Position and velocity are restricted so that $-1.2 \leq p \leq 0.5$ and $-0.07 \leq v \leq 0.07$. The goal is considered attained when the car reaches the position of 0.5.

3.2.2 Q-Learning Results

We demonstrate Q-learning using this new kernel formulation on the mountain car problem. Furthermore, we use the continuous action space $\mathcal{A}_{cont} = [-1, 1]$. For simplicity, to compute $\gamma \max_a Q(x_{n+1}, a)$, we evaluate $Q(x, a)$ at the values $\{0, -.9, -.8, \ldots .8, .9, 1\}$, although there are methods, such as the mean shift algorithm [47] [48], to compute the maxima that do not have the limitations of evaluating on a grid. The state kernel size σ_x was set to 0.12.

First, we compare how action kernel size affects learning. Table 3-2 shows the number of steps to reach the goal state starting from at rest on the hill while starting

Table 3-2. Number of steps from cart at rest at bottom of hill to goal for various action kernel sizes

action ks	0.01	0.1	0.25	1.0	5.0	rand
#steps	8759	3634	2299	1138	420	41994

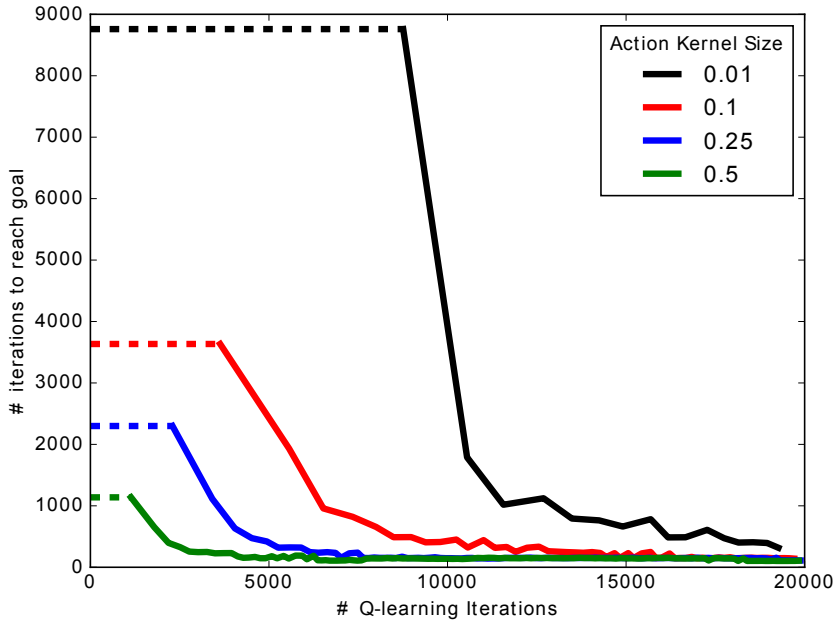

Figure 3-2. Number of steps required to complete the mountain car problem for different action kernel sizes evolving over 20000 iterations. Each time the mountain car reaches the goal, its state is reset to the -0.5 position and 0 velocity. The number of steps is measured by subtracting the goal iteration number from the reset iteration.

from at rest at the bottom of the hill. Furthermore, Figure 3-2 shows the number of iterations required to reach the goal while training Q-learning policy online. After the mountain car reached the goal state, it was reset to the initial position and training continued. The action kernel sizes of $\sigma_a = 1.0$ and $\sigma_a = 5.0$ are not shown because they largely overlap the plot for $\sigma_a = 0.5$.

Clearly the optimal policy is learned more quickly with larger action kernel sizes. This is because there is no advantage, for this problem, in ever taking an action that does not move the cart with maximum force. That is, the fastest solution uses only the actions in the set $\{+1, -1\}$, while avoiding all other actions in between.

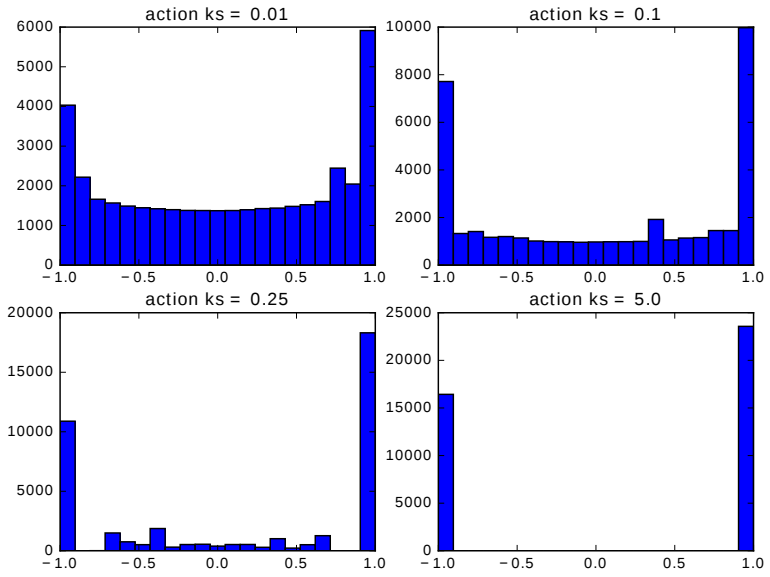

Figure 3-3. Histogram of actions chosen over 40000 iterations for four different kernel sizes

For each Q-learning update, a very large action kernel associates the update with all actions $a \in A$ strongly. This has the effect of forcing the selected actions to the boundaries of the action set. Figure 3-3 shows a histogram of the actions selected by the Q-learning algorithm for four different kernel sizes while training over 40000 iterations. This phenomenon can clearly be seen as the action kernel size grows. For a small action kernel size, actions are explored uniformly across the interval $[-1, 1]$, until the algorithm learns to associate the most value with the actions $\{+1, -1\}$. For very large action kernel sizes, the only actions taken are $+1$ and -1.

Furthermore we compare different parameter choices while using experience replay. Unlike other implementations, we find that updating the latest result along with the experience replay batch provides better results. This is most likely because rare events, such as reaching the top of the hill in the mountain car problem, have a 100% chance of being experienced at least once.

We vary two parameters—modulo and batch size. The modulo parameter specifies at what iteration multiple the q-model is batch updated by experience replay. That is, if modulo = M, then the q-model is batch updated every M iterations. The batch size is the number of experiences used for each update as described by Equation (3–11). Figure 3-4 shows the number of iterations required to reach the goal while training Q-learning policy online for different modulo and batch size settings. For this experiment we used a learning rate $\alpha = 0.5$, state kernel size $\sigma_x = 0.12$, action kernel size $\sigma_a = 0.25$, and discount factor $\gamma = 0.99$.

Comparing the Figure 3-4 plots, the modulo parameter of experience replay strongly affects the convergence time to the optimal policy. That is, updating the q-model from memory more often leads to faster convergence.

On the other hand, there is little difference between the plots of the same modulo (rows) and different batch sizes (columns). This can be explained by recalling Equation (3–12) rewritten here including batch size normalization for convenience:

$$\omega_i = \omega_{i-1} + \frac{\alpha}{B} \sum_{n \in I} e_n \varphi(x_n)$$

The weight vector in the RKHS is updated by averaging the weighted sum of the feature vectors $\varphi(x_n)$, where $\|\varphi(x_n)\| = 1$. Thus the norm of the change in weight vector is less than or equal to the average of the magnitude of the error

$$\| \sum_{n \in I} e_n \varphi(x_n) \| \leq \sum_{n \in I} \| e_n \varphi(x_n) \| = \sum_{n \in I} |e_n| \tag{3–16}$$

Thus increasing the batch size cannot increase the expected norm of the weight change $E\|\Delta\omega\|$. Thus for appropriately small updates, batchsize will not speed convergence. Apparently, for this problem and learning rate, a small batch size is sufficient to provide appropriate updates for the model. However, for larger learning rates, this may not be the case.

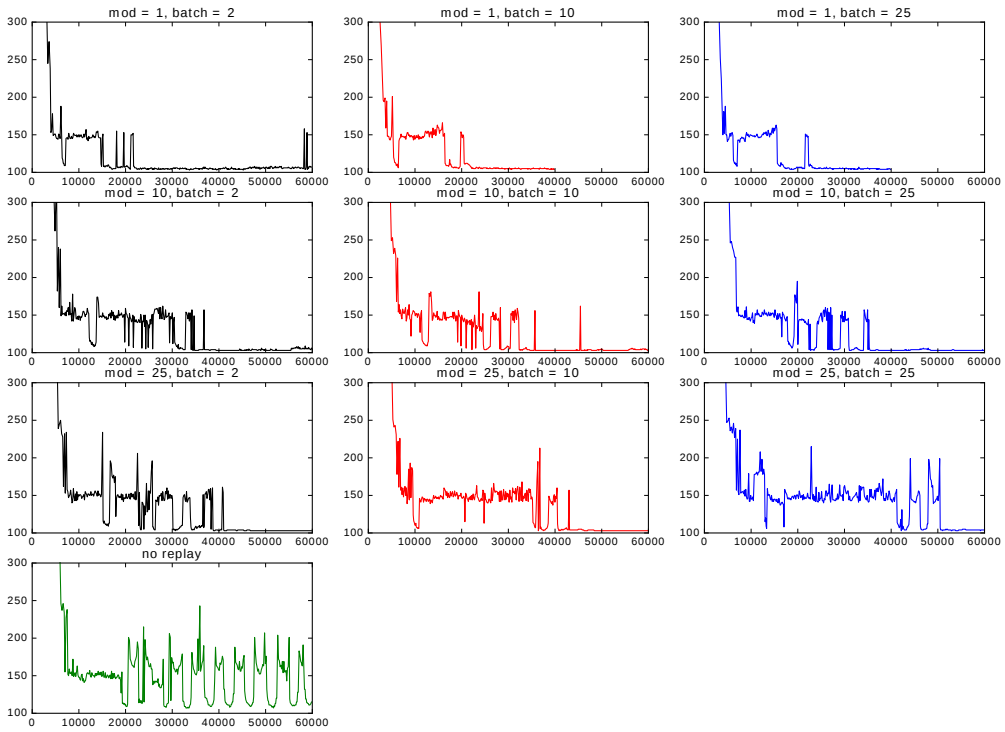

Figure 3-4. Number of steps required to complete mountain car problem for different experience replay modulo and batch size evolving over 60000 iterations. Each time the mountain car reaches the goal, its state is reset to the -0.5 position and 0 velocity. The number of steps is measured by subtracting the goal iteration number from the reset iteration. Modulo represents how often experience replay is applied. Batch(size) is the number of experiences replayed each application.

To show this, we re-ran the same experiment with learning rate $\alpha = 3.0$ and modulo 10. We compare learning curves for batch sizes of 2 and 4. Figure 3-5 shows that for this learning rate, the learning curve for a batch size of 2 begins to diverge at approximately iteration $20,000$. The learning curve for batch size 25, on the other hand, converges very quickly. Comparing this curve with its corresponding curve in 3-4, we find that this learning rate achieves convergence at the optimal policy much more rapidly.

Figure 3-6 shows the estimated value of each state $x \in X$, i.e., the expected sum of future rewards $E\{\sum_i^\infty r_i\} = \max_a Q(x, a)$ from each state, for each batch size after

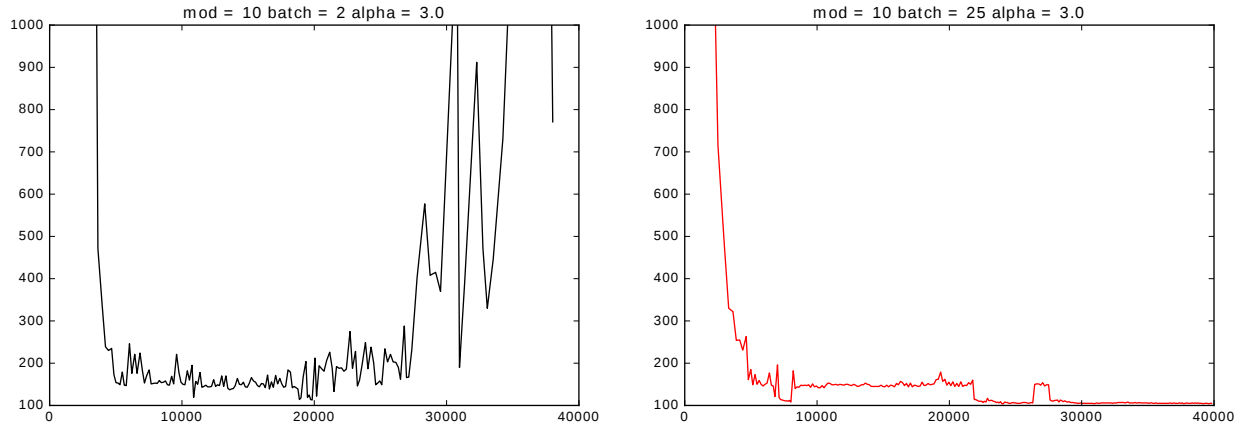

Figure 3-5. Number of steps required to complete mountain car problem for a large learning rate and two different batch sizes (batch sizes of $B = 2$ and $B = 25$). Each time the mountain car reaches the goal, its state is reset to the -0.5 position and 0 velocity. The number of steps is measured by subtracting the goal iteration number from the reset iteration. The parameters modulo $M = 10$ and learning rate $\alpha = 3.0$ were used. The left plot corresponding to $B = 2$ begins to diverge at around 20000 iterations.

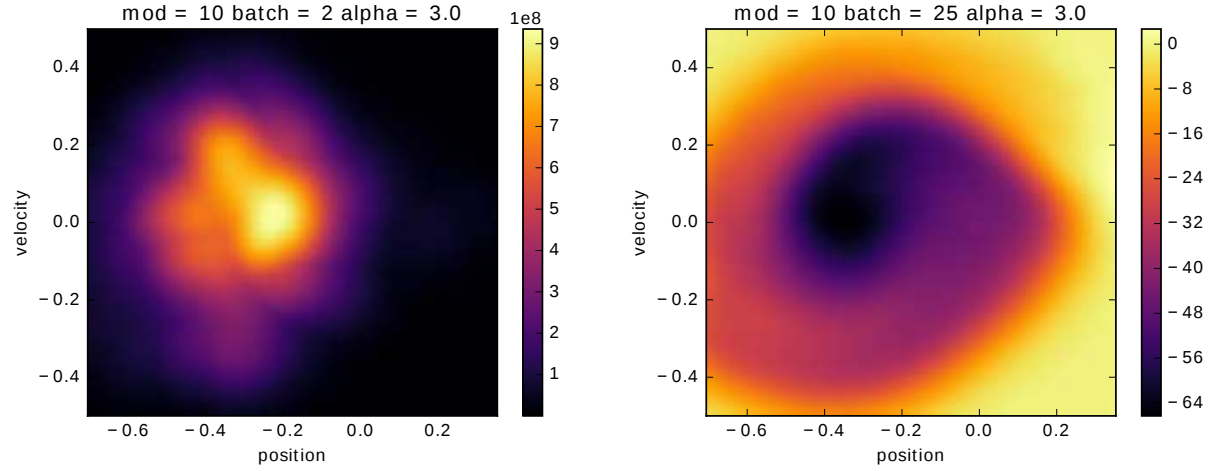

Figure 3-6. Heat map for the value estimate of each state in the mountain car state space $x \in X$ for two different batch sizes after training for $40,000$ iterations. The parameters modulo $M = 10$ and learning rate $\alpha = 3.0$ were used. The left plot corresponding to $B = 2$ shows that the value function estimate has clearly diverged, with the highest value estimate of approximately 9×10^8.

training for $40,000$ iterations. The plot correspoding to a batch size of 2 has clearly diverged. Notice the scale of the color bar; some states achieved a value estimate of 10^8. The value estimate plot for a batch size of 25, on the other hand, produces a spiral corresponding to starting at position -0.5 and velocity 0.0, climbing the hill first to the right, then to the left, and then finally to reaching the goal on the right.

Thus, we have more fully developed the Q-KTD algorithm by augmenting it with the action kernel and experience replay. We have shown that the action kernel allows us to generalize experiences across actions using the mountain car problem as a test bed. We have further shown that experience replay in the RKHS is extremly effective and fast way to improve the speed of learning an optimal policy using Q-KTD.

CHAPTER 4
REINFORCEMENT LEARNING WITH METRIC LEARNING

The goal of learning is to teach an agent to produce the optimal set of actions in a given environment. In the context of Markov Decision Processes [49], this is often defined as selecting the sequence of actions leading to the greatest average cumulative reward. Learning to exploit real-world scenarios is nearly always plagued by the curse of dimensionality. That is, when the "world" the data exists in is represented by many variables, the problem becomes so large that the agent requires an unreasonable amount of experience to test actions in all relevant scenarios. This motivates finding efficient representations of the data using a process known as "feature selection." Although feature selection [50] is well studied in other machine learning tasks, such as classification, in RL features are often hand-selected based on expert domain knowledge. A typical work-around to the curse of dimensionality involves exploiting previous experiences when a similar unknown scenario presents itself. In the context of reinforcement learning, there have been a number of different methods proposed and employed to exploit previous experience. These include k-nearest- neighbor, linear interpolation, and local weighted averaging. Gordon [51] proved that interpolators that are non-expansion mappings such as k-nearest-neighbor are stable function approximators.

A (discounted) MDP can be represented mathematically as a 5-tuple (X, A, P, R, γ), where X is the set of all possible states (state space), A is the set of all possible actions (action space), P is the set of transition probabilities, R is the set of rewards, and γ is the discount factor. The value function [25] [26] [27] is defined as the sum of expected future discounted rewards, starting in state x and performing action a:

$$Q^{\pi}(x, a) = E\left[\sum_{t=0}^{\infty} \gamma^{t} R_{t+1} | X_0 = x, A_0 = a\right], 0 \leq \gamma < 1 \tag{4–1}$$

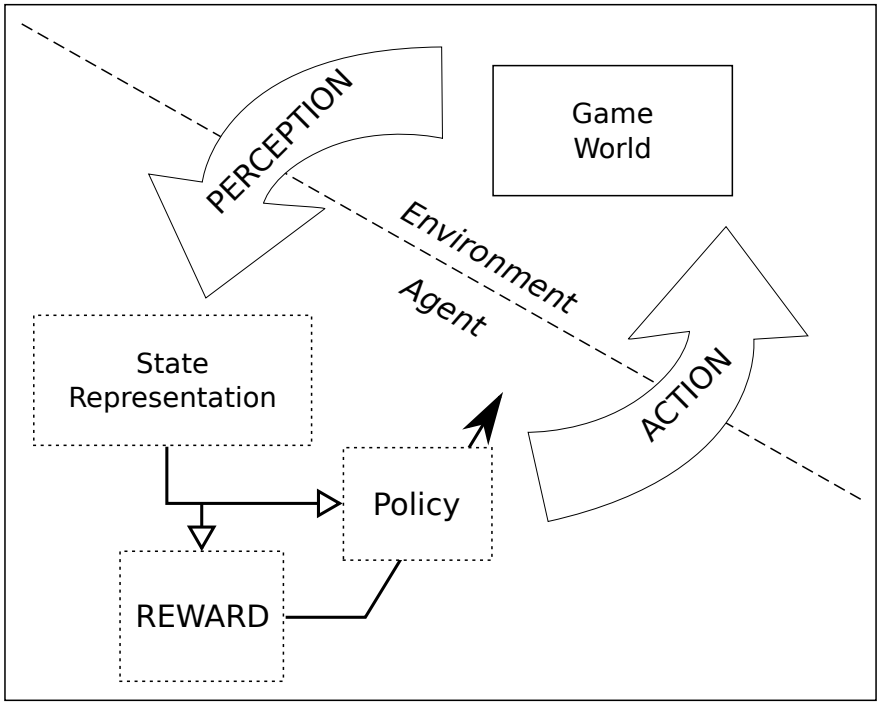

Figure 4-1. The perception-action-reward cycle describes the processing of an agent playing a game. The player forms an internal state representation of the gaming environment through perception. Based on the current game state the player's policy determines which action to take. After each action, or inaction, the gaming environment transitions from its current state to the next, yielding a reward. The player modifies the policy based on the observation of this transition.

It is intractable to calculate the value functions of MDPs directly because it requires expectations over all future times. Using stochastic approximation, (tabular) Q-learning is able to approximate the value function iteratively using the following recursive update:

$$Q(X_t, A_t) \leftarrow Q(X_t, A_t) + \alpha\delta \tag{4–2}$$

$$\delta = R_{t+1} + \gamma max_a Q(X_{t+1}, a) - Q(X_t, A_t) \tag{4–3}$$

Where α is the learning rate, and $Q(\cdot, \cdot)$ is the current Q-function estimate. The initial stages of Q-learning can be very difficult, as actions must be made based on poorly estimated quantities. For tabular Q-learning, random actions must be made when there is no prior experience for that state. Function approximation methods are capable of interpolating Q-values when new states are encountered, but this may take some time

Table 4-1. Q-learning variant with nearest neighbor action selection

6a)	if $\max_a Q(x, a)$ exists, then:
6b)	$a \leftarrow \text{argmax}_{a'} Q(x, a')$
6c)	else:
6d)	$x_0 \leftarrow$ nearest neighbor to x
6e)	$a \leftarrow \text{argmax}_{a'} Q(x_0, a')$

before producing reliable quantities. To mitigate these effects, we use the Q-values from nearest neighbor states to serve as surrogates for action selection and as targets for Q-learning updates when data is unavailable for new states.

Nearest neighbor regression is a simple method for interpolating the value of a function given a set of input and output examples. Consider the set of points $\{x_k, f(x_k)\}_{k=1}^N$, where $x_k \in \mathcal{X}$ and $f : \mathcal{X} \to \mathcal{Y}$. For any $x \in \mathcal{X}$ we can approximate the value of $f(x)$ as $f(x_n)$ where $x_n = \text{argmin}_{x_k} d(x, x_k)$ and $d(\cdot, \cdot)$ is a distance function.

Suppose states $\{x_k\}_{k=1}^N$ have been visited, thus there exist table entries for the Q-values of these states at the actions that were taken in them. For step 6 of Table 2-1, if an entry for x in $Q(x, a)$ does not exist, typically an action is selected randomly. Using nearest neighbor interpolation, we can instead use the Q-values of the nearest recorded state. This helps guide actions early in learning so that when only a few points are recorded, action selection is not completely random. Table 4-1 shows a modified step 6 to reflect this modification.

Furthermore Q-values can be updated using nearest-neighbor interpolation. That is, Equation (2–10) can be modified so that if X_{t+1} has never been visited, the nearest neighbor to X_{t+1} can be used. Table 4-2 gives an updated Q-learning algorithm which includes using nearest neighbor for both action selection and update.

Our approach relies on the distance function d to provide a measure of similarity between states. Typically, nearest neighbor is performed using the Euclidean metric, i.e. if x and x' are feature vectors, and x_i represents the ith element of x, then $d(x, x') = \sqrt{\sum_i (x_i - x'_i)^2}$. However, not all features of the state are equally important, and if less

Table 4-2. Q-learning using NN with ϵ-greedy exploration

1)	Initialize parameters γ, α, ϵ
2)	Repeat:
3)	$\epsilon_0 \leftarrow$ random number between 0 and 1
4)	$x \leftarrow$ current state
5)	if $\epsilon_0 > \epsilon$ then:
6a)	if $\max_a Q(x, a)$ exists, then:
6b)	$a \leftarrow \text{argmax}_{a'} Q(x, a')$
6c)	else:
6d)	$x_0 \leftarrow$ nearest neighbor to x
6e)	$a \leftarrow \text{argmax}_{a'} Q(x_0, a')$
7)	else:
8)	$a \leftarrow$ random action
9)	Take action a, observe state transition
10)	$x' \leftarrow$ new state
11)	$r \leftarrow$ reward associated with state transition
12a)	if $\max_a Q(x', a)$ exists, then:
12b)	$\delta \leftarrow r + \gamma \max_{a'} Q(x', a') - Q(x, a)$
12c)	else:
12d)	$x'_1 \leftarrow$ nearest neighbor to x'
12e)	$\delta \leftarrow r + \gamma \max_{a'} Q(x'_1, a') - Q(x, a)$
13)	$Q(x, a) \leftarrow Q(x, a) + \alpha \cdot \delta$

important features dominate the Euclidean distance, then neighboring states may not be similar relative to the task at hand. It is therefore important that we use *metric learning* to learn a weighted Euclidean metric. That is, we learn a weight vector w, such that $d_w(x, x') = \sqrt{\sum_i (w_i(x_i - x'_i))^2}$. The weights can be seen as weighting the importance of features.

As seen in Table 4-2, the algorithm utilizes nearest neighbor interpolation to select actions and to update Q-values.

4.1 Metric Learning for Automatic State Representation

For high-dimensional data, the problem of mapping state-actions to value functions is intimately related to the problem of selecting a feature mapping. While many feature variables may be relevant to the decision process, others may be effectively extraneous. The presence of these "noisy" features greatly impedes learning because RL relies on revisiting states. Irrelevant features cause states which may be inherently similar to appear different. For instance, consider the task of catching a baseball during the day, and catching that same baseball at night. While the task and optimal actions are identical, standard RL fails to recognize these situations as the same because the night and day backgrounds are significantly different. Feature selection consists of including, excluding, weighting, or combining features to make a new set of features. Typically, feature selection is a preprocessing step for RL. As an alternative to feature selection, since nearest-neighbor regression (and other regression methods) rely on an underlying metric, we directly modify the distance function to improve regression. With the proper criterion, the weighting in the distance function can be learned to produce the desired representation. Intuitively, we wish to maximize the mutual information or dependence between the representation and the goal. Note that the goal is encoded in the Q-function values themselves, i.e. if the Q-functions are similar for two states, then these states yield similar outcomes with respect to the cumulative rewards. Thus, the objective is to maximize the dependence between the state space representation and the Q-function evaluations. To do this, we attempt to maximize the correlation between two (metric-dependent) kernel functions, with a measure known as centered alignment [52] [53]. That is, we attempt to alter the metric on the state space so that if $x, y \in X$ are similar, then $Q(x, a), Q(y, a)$ will be similar as well.

Since nearest neighbor regression relies on the distance function, we directly modify the distance function to improve the regression as an alternative to feature selection. Good features for regression are ones whose similarity/dissimilarity implies

similarity/dissimilarity in the dependent variable. In the case of reinforcement learning, the state representation is the regressor and the Q-value is the dependent variable.

For linear regression the relationship between the independent and dependent variables amounts to correlation, but more generally it can be quantified by statistical dependence [54]. We use a metric learning algorithm named "Centered Alignment Metric Learning (CAML)" [53] that uses a kernel-based statistical dependency measure [52] as the objective function.

Our objective for metric learning is to maximize the dependence between the state space representation and the Q-function evaluations. Centered kernel alignment is used to evaluate the statistical dependence in terms of kernel functions. Kernel functions are bivariate measures of similarities. The dependence between random variables is measured by how often similarity in one variable corresponds to similarity in the other variable. Somewhat confusingly this can be stated as "how similar are kernel evaluations (a measure of similarity) for the two variables?" Fortunately, this has a clear answer: just like the correlation coefficient for pairs of real valued variables, a natural measure of similarity between these kernel functions is the expected value of their centered and normalized inner product.

Let $\kappa_x(.,.)$ and $\kappa_y(.,.)$ denote the kernel functions defined in $\mathcal{X} \times \mathcal{X}$ and $\mathcal{Y} \times \mathcal{Y}$, respectively. Centered alignment is written as

$$\rho_{\kappa_x,\kappa_y}(x,y) = \frac{\mathsf{E}_{x,y}\mathsf{E}_{x',y'}[\tilde{\kappa}_x(x,x')\tilde{\kappa}_y(y,y')]}{\sqrt{\mathsf{E}_x\mathsf{E}_{x'}[\tilde{\kappa}_x^2(x,x')]\mathsf{E}_y\mathsf{E}_{y'}[\tilde{\kappa}_y^2(y,y')]}} \tag{4–4}$$

$$\begin{aligned}
\tilde{\kappa}_i(z,z') &= \langle \tilde{\phi}_i(z), \tilde{\phi}_i(z') \rangle \\
&= \langle \phi_i(z) - \mathsf{E}_z[\phi_i(z)], \phi_i(z') - \mathsf{E}_{z'}[\phi_i(z')] \rangle \\
&= \kappa_i(z,z') - \mathsf{E}_{z'}[\kappa_i(z,z')] - \mathsf{E}_z[\kappa_i(z,z')] \\
&\quad + \mathsf{E}_{z,z'}[\kappa_i(z,z')].
\end{aligned} \tag{4–5}$$

Where x', y', z' are random variables independent and identically distributed to the random variables x, y, z.

The centered alignment $\rho_{\kappa_x,\kappa_y}(x, y)$ is a measure of statistical dependence between x and y. For positive-definite-symmetric kernels, $\rho_{\kappa_x,\kappa_y} \in [0, 1]$ [52]. Centered alignment is essentially a normalized version of the Hilbert-Schmidt Information Criterion [55].

Given a dataset $\{x_i, y_i\}_{i=1}^n$, define kernel matrices X and Y, with the elements at i, j being $X_{ij} = \kappa_x(x_i, x_j)$ and $Y_{ij} = \kappa_y(y_i, y_j)$. An empirical estimate of the centered alignment can be computed directly from these kernel matrices as

$$\hat{\rho}(X, Y) = \frac{\langle \tilde{X}, \tilde{Y} \rangle}{\sqrt{\langle \tilde{X}, \tilde{X} \rangle \langle \tilde{Y}, \tilde{Y} \rangle}} = \frac{\langle \tilde{X}, \tilde{Y} \rangle}{\|\tilde{X}\|_2 \|\tilde{Y}\|_2} \tag{4–6}$$

where $\tilde{X}$ and $\tilde{Y}$ are the *centered* kernel matrices. The centered kernel is computed as

$$\tilde{X}_{ij} = X_{ij} - \frac{1}{n}\sum_{a=1}^{n} X_{aj} - \frac{1}{n}\sum_{b=1}^{n} X_{ib} + \frac{1}{n^2}\sum_{a=1}^{n}\sum_{b=1}^{n} X_{ab}. \tag{4–7}$$

Using matrix multiplication, $\tilde{X} = HXH$, where $H = I - \frac{1}{n}\mathbf{1}\mathbf{1}^\top$ is the empirical centering matrix, I is the $n \times n$ identity matrix, and $\mathbf{1}$ is a vector of ones. The computational complexity of the centered alignment between two $n \times n$ kernel matrices is $\mathcal{O}(n^2)$.

We now consider centered alignment as an objective for explicitly optimizing the coefficients of the weighted Euclidean distance on the space of the independent variable x. If we use the Gaussian kernel, the weights w are present in the argument of the exponential,

$$\begin{aligned} k_w(x, x') &= \exp(-d_w(x, x')^2) \\ &= \exp\left(-\sum_i w_i (x_i - x_i')^2\right). \end{aligned}$$

Any kernel size is absorbed into the weights, so an additional parameter is unnecessary. The unweighted Gaussian kernel is used on the dependent variable

$$k_y(y, y') = \exp\left(-\frac{1}{2\sigma^2} d(y, y')^2\right),$$

as we are not learning a weighting on this space.

As an optimization technique, we use stochastic gradient ascent over small batches taken randomly from the set of available examples. The optimization of the centered alignment over the weights can be stated as

$$\underset{w \geq 0}{\text{maximize}} \;\; \log(\rho_{\kappa_w, \kappa_y}(x, y)), \tag{4–8}$$

where we take the logarithm of ρ to simplify calculation of the gradient. Taking the logarithm does not change the global optima, and using the logarithm of ρ as the objective resulted in better performance empirically.

When the empirical estimate of centered alignment is substituted the explicit objective function is

$$\begin{aligned} f(w) &= \log(\hat{\rho}(X_w, Y)) \\ &= \log(\text{tr}(X_w H Y H)) - \log(\sqrt{\text{tr}(X_w H X_w H)}) \\ &\quad - \log(\text{tr}(Y H Y H)) \end{aligned} \tag{4–9}$$

where X_w is the kernel matrix of the independent variables (with the dependence on the weighting present in the kernel k_w made explicit) and Y is the kernel matrix of the dependent variables.

For Q-learning with metric learning, we find a weighting on the state variables to maximize the dependence between the weighted state and the Q-values for those states, under the same action. In other words, two states are similar if performing the same action leads to similar outcomes. This requires a set of visited states and their approximate Q-values, acquired from the initial stages of Q-learning without any feature

weighting. Consider the subset $\{x_i, \hat{Q}(x_i, a)\}_{i=1}^{n_a}$ of state-value pairs for a specific action, a. Let $f_a(w)$ be the log of the empirical estimate of centered alignment for this set, defined by Equation ((4–9)). A new objective function can be constructed, which takes all actions into account:

$$f(w) = \sum_{a \in \mathcal{A}} f_a(w). \tag{4–10}$$

Finding the weights w which maximize Equation (4–10), results in an improved feature weighting. Using this weighting, improved Q-values can be estimated, which can then be used to learn a new weighting in an iterative fashion.

4.2 Reinforcement Learning for Games

Learning to successfully solve or "win" a game is the process of discovering the right actions to achieve goals or subgoals. As players learn to play a game they begin to associate actions or sequences of actions with particular game scenarios. Gameplay can be considered a decision process in which a player must decide, based on his current knowledge of the game world, what actions or sequences of actions will best lead to the game's objective. More generally, the player learns a policy that maps any given game scenario to an action permitted by the game world.

Traditional artificial intelligence methods, such as finite state machines, hard code policies based on expert knowledge of the game and its goals. Such a policy is unable to adapt to changing circumstances. Furthermore, the policy is limited by the knowledge of the AI designer.

Reinforcement learning (RL) [26, 27] is a natural paradigm for a machine to mimic the biological process of learning to complete objectives. It is applied to sequential decision problems, in which the outcome of an action may not be immediately apparent. With RL, credit can be assigned to the actions leading to an outcome, which provides feedback in the form of a reward. The feedback that occurs in response to actions is combined with previous experience. Thus, RL serves as a principled means of learning

policies for decision making directly from experience. RL has been successfully applied to games, most famously with Tesauro's world-class backgammon player, TD-gammon [10]. RL has also been applied to games such as Tetris [27] and chess [56, 57].

Most approaches described in the references above are limited in their application to other games. While RL agents learn their policies and are thus not tied to any particular application, this generality of RL has not been effectively exploited for games. This is the result of the difficulties in choosing the relevant variables to represent a game scenario for the RL algorithm. If an agent is not given enough information about the state of the game, the results of actions cannot be attributed to the proper cues from the game, and thus the agent will never learn. However, when game scenarios are represented with many variables, the problem often becomes so large that the agent requires an unreasonable amount of experience to test actions in all important game scenarios. This problem is known as the "curse of dimensionality" [25], and it motivates finding efficient representations of game scenarios, or "feature selection". While feature selection is well-studied for other machine learning tasks, such as classification [50], in RL it is often done heuristically or with expert knowledge [26].

Even with careful feature selection, there are still too many game scenarios for an agent to exhaustively explore all possibilities. When a previously unseen scenario arises, it is important to exploit previous experience by interpolating amongst similar scenarios [58] [51]. For our experiments, we use the simple nearest-neighbor interpolation method for its simplicity in both implementation and computation. Previous examples of nearest-neighbor interpolation being used with RL methods include [59] and [60]. Gordon [51] proved that for fitted dynamic programming and RL methods, k-nearest-neighbor is a stable function approximator. The appropriate choice for this interpolation turns out to be highly dependent on the way a game scenario is represented. Thus, we will look at various representations and their effect on the ability of RL to learn in terms of learning speed and goodness of the final policy. We will use a (Python) clone

of the classic 1981 arcade game, “Frogger,” as shown in Figure 4-2, as our test bed. Similar work on another Frogger clone has been performed by Cobo [61], using learning from demonstration (LfD) and state abstraction combined with RL methods. Wintermute [62] experimented with another similar game, “Frogger II,” using imagery to predict future states and Q-learning to assign a value to abstracted states.

4.3 Results: Dimensionality Reduction in Visual Task

To demonstrate the ability of our metric learning approach to reduce the size of the state representation, we test the algorithm on the arcade game Frogger. Video games serve as an excellent test bed for RL in visual systems since they are simple to implement, yet provide a challenging task in a complex and interactive environment. The action set in a game of Frogger includes movement up, down, left, and right. The goal of the game is to maneuver the frog from the bottom of the screen, safely to the top, avoiding cars and crossing the river using moving turtles and logs. While computer vision techniques can be used to identify the separate objects present on the screen, the agent does not known how these objects relate to the chosen actions and the outcome of the game. Providing the agent with a catalog of the objects and their locations is overwhelming. Metric learning for RL can be used to identify the relevant objects.

We constructed three different feature sets which we will label “holistic,” “local,” and “irrelevant.” The “holistic” feature set contains the location of the frog and every sprite in the game. The “local” feature set contains the location of the frog and the location of sprites in the frog’s immediate vicinity. For the holistic features, features 3-7 are the car sprites x-locations. Features 8-12 are the turtle and log sprite’s x-locations. Features 13-17 indicate whether the corresponding home location is empty or filled. For the local features, features 1-4 and 6-9 indicate whether a sprite is in the corresponding square-shaped box. Feature 5 is the y-location of the frog. Not pictured are features 10-14, which (like the holistic feature construction method) indicate whether the home locations are empty or filled. For the “irrelevant” feature set, features 1-4 and 6-12

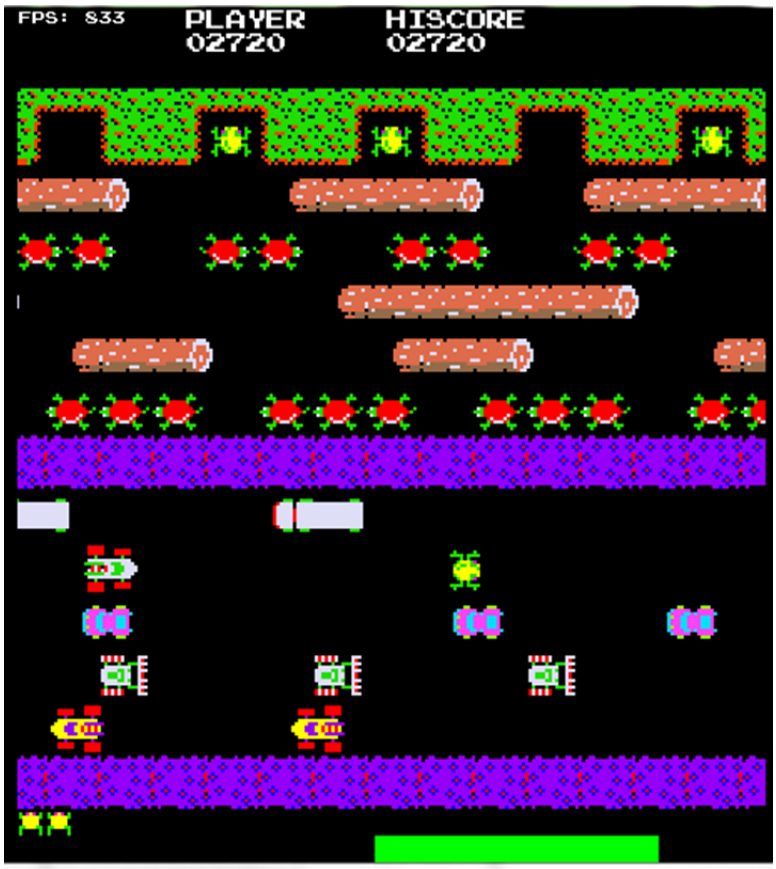

Figure 4-2. Screenshot of Frogger game. In a normal game, the player controls the frog using a joystick or keyboard. The only controls are up, down, left, and right. The goal of the game is to maneuver the frog to the five home positions (shown at the top of the screen), without allowing the frog to be hit by one of the cars (bottom half of screen), or drown in the river (top half of screen). The player navigates the river by jumping on the logs and turtles. Each time the player maneuvers the frog to the one of the home positions, a new player-controlled frog is respawned at the bottom of the screen. Once all five home positions are filled, a new level begins and process is repeated. Pitfalls for the player to watch out for include turtles periodically diving underneath the water, and a crocodile that fills one of the home positions at random intervals.

indicate whether a sprite is in the corresponding square shaped box and feature 5 is the y-location of the frog. Features 13-17 indicate whether the home locations are empty or filled. Clearly, features 10-12 are less relevant in determining the action the agent needs to take. Since each frog jump is 32 pixels, we discretized the x- and y- position information to multiples of 32. We use the following reward structure: +1 for the frog reaching a home, -1 for the frog dying, and +5 for completing a game by reaching all five homes. Furthermore, to speed up learning, a small positive reward of +0.1 was given for reaching the midpoint and a small negative reward of -0.1 was given for moving down from the midpoint.

We evaluate five variants of the Q-learning algorithm based on the speed of learning a policy. For each experiment, we set the discount factor γ to 0.9, the learning rate α to 0.1, and the exploration parameter ϵ to 0.1. These values were chosen based on experimentation and knowledge from previous Q-learning applications. The first variant is the classical tabular Q-learning algorithm. The next two, which we label "NN- No weights" and "NN - Weighted" use nearest neighbor action selection. The final two use nearest neighbor action selection as well as the nearest neighbor update as shown in Table 4-2. The weighted variants weight features according to weights as determined by CAML. The unweighted variants simply weight each feature equally with a weighting of one. To determine the weights for each feature we first ran the classical Q-learning algorithm to obtain a table of states and their corresponding Q-function estimates. We then ran the CAML algorithm on this data to learn the weighting.

4.3.1 Holistic Features

We ran each Q-learning variant with holistic features for 200,000 episodes. Figure 4-10 shows the cumulative number of games won versus the number of episodes spent learning. Since using the algorithm shown in Table 4-2 is so much more successful than the other variants, we show only the first 60,000 episodes in the bottom view. It is apparent that the standard tabular Q-learning algorithm fails to learn an improved

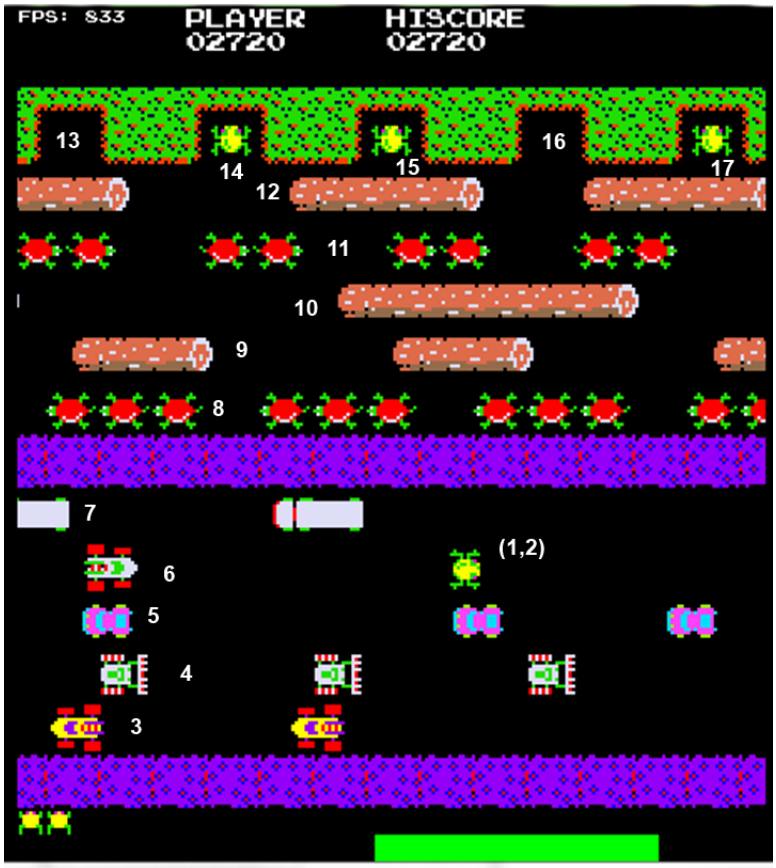

Figure 4-3. Visualization of “holistic” feature construction. The first and second features are the x- and y-locations of frog respectively. Features 3-7 are the car sprites x-locations. Features 8-12 are the turtle and log sprite’s x-locations. Features 13-17 indicate whether the corresponding home location is empty or filled.

policy. This occurs because the state space is so large that states are rarely visited more than once, and as a result most actions must be random ones. The unweighted nearest neighbor variants perform better because action can be selected based on the experience of nearby states even when the current state is new.

Figure 4-6 shows weights learned for each feature. It should be noted that the frog x- and y- locations are weighted higher than the rest of the features. Note the x-position is particularly important because it contains the information rquired for the frog to know when to move up. Weighting these two features makes intuitive sense, because the configuration of the car sprites is irrelevant if the frog is in a completely different position.

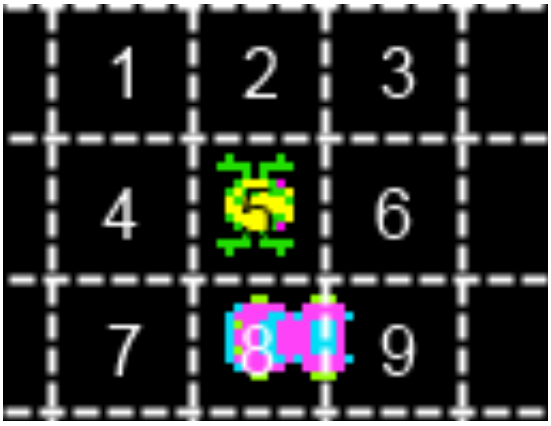

Figure 4-4. Visualization of “local” feature construction. Features 1-4 and 6-9 indicate whether a sprite is in the corresponding square-shaped box. Feature 5 is the y-location of the frog. Not pictured here are features 10-14, which (like the holistic feature construction method) indicate whether the home locations are empty or filled.

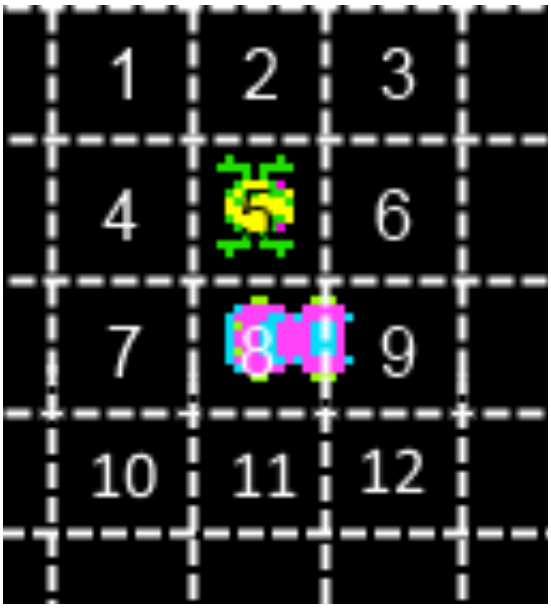

Figure 4-5. Visualization of “local” feature construction with irrelevant features. Features 1-4 and 6-12 indicate whether a sprite is in the corresponding square-shaped box. Feature 5 is the y-location of the frog. Features 10-12 are less relevant in determining the action the agent needs to take. Not pictured here are features 13-17, which indicate whether the home locations are empty or filled.

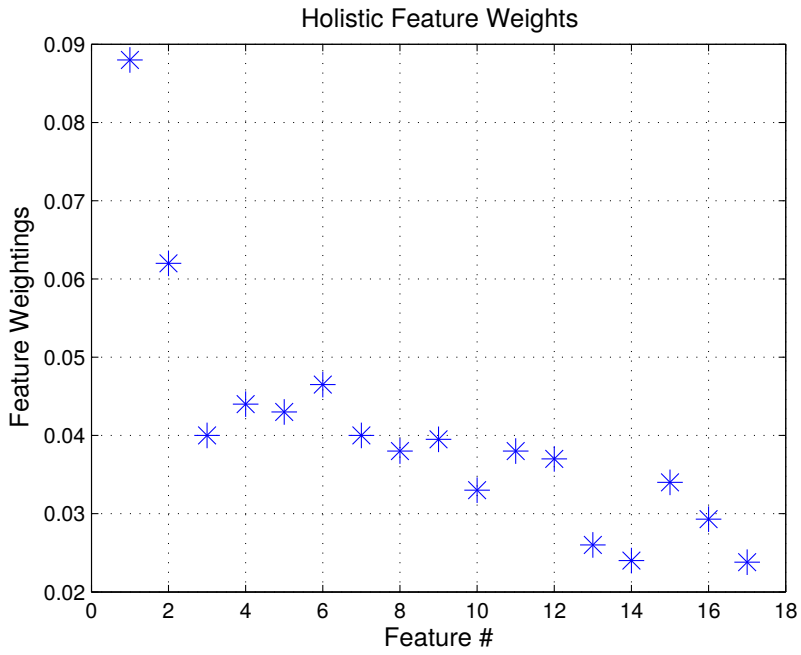

Figure 4-6. (Holistic Features) Feature weightings learned using state and Q-value data and CAML. The first and second features are the x- and y- locations of frog respectively. Features 3-7 are the car sprites x-locations. Features 8-12 are the turtle and log sprite's x-locations. Features 13-17 are indicate whether the corresponding home location is empty or filled.

The "home" features are given the least weight, implying that the optimal action for most states is not dependent on which combination of homes are filled. For example, the moving upwards is a good starting move regardless of where the frog needs to be in the distant future. This weighting provided by metric learning is similar to one that was hand-selected for good performance experimentally.

Figures 4-9 and 4-10 clearly show that using either or both weighted features and nearest neighbor updates improves the speed of learning. The weights ensure that two states which lead to similar outcomes are themselves similar with respect to our metric, while states that lead to different outcomes are far away. This helps in selecting a suitable action when no "best" action is known. The nearest neighbor update allows previously unvisited states to quickly take on the Q-values of similar states.

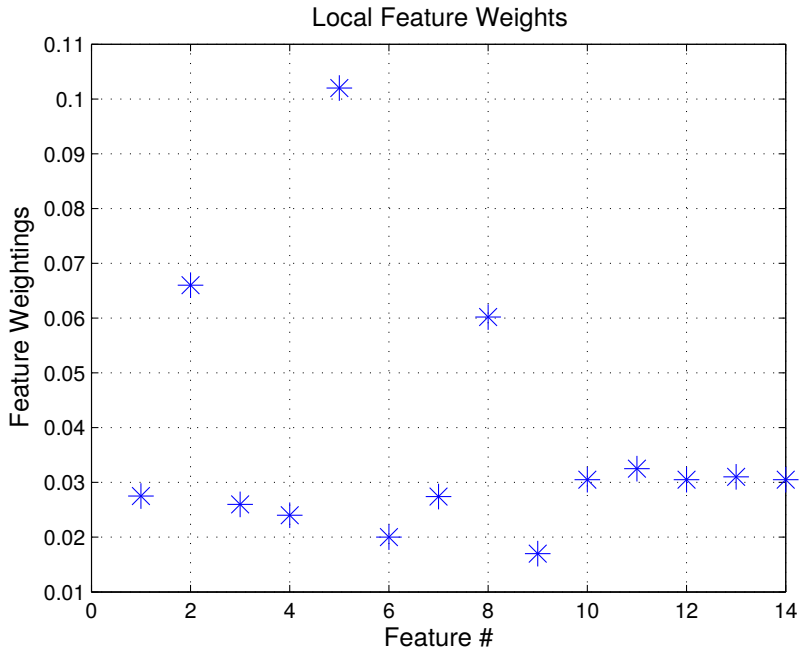

Figure 4-7. (Local Features) Feature weightings learned using state and Q-value data and CAML. Features 1-4 and 6-9 indicate whether a sprite is in the corresponding square-shaped box. Feature 5 is the y-location of the frog. Features 10-14 indicate whether the home locations are empty or filled.

4.3.2 Local Features

We ran each Q-learning variant with local features for 200,000 episodes. Figures 4-11 shows the number of games won versus the number of episodes. The difference in results between Q-learning variants for local features is much smaller than for holistic features because the local feature state space is much smaller. Even so, using weighted features or the algorithm presented in Table 4-2 boosts learning by a noticeable amount.

We do not show weighted features with the algorithm in Table 4-2 because that variant's results are almost identical to the unweighted case. Weighted versus unweighted is much less significant for this feature set because, unlike the holistic feature set, there are no variables that are significantly more important than the others for determining the correct action.

Figures 4-7 gives a visualization of feature importance, as learned by CAML, for the local features. It can be seen that the most highly-weighted feature is the frog's

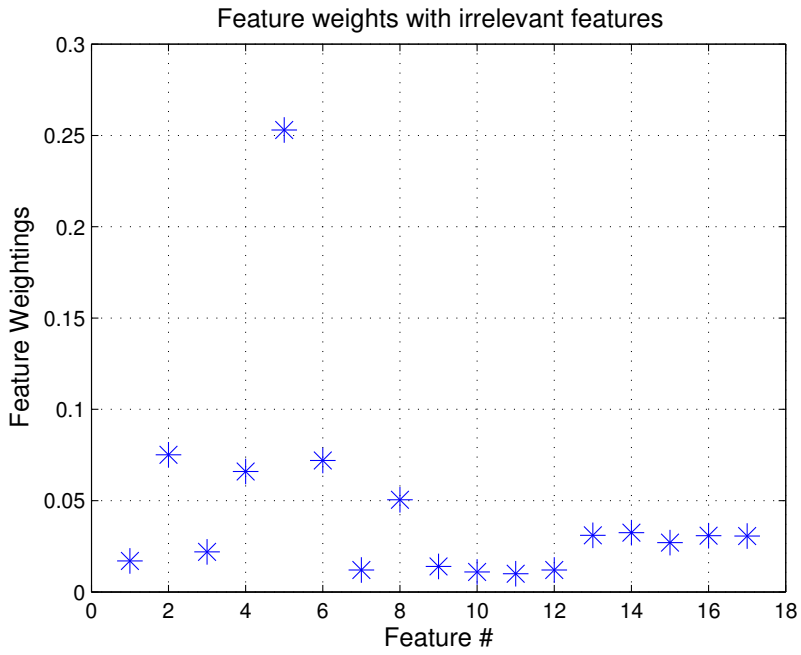

Figure 4-8. (Local + Irrelevant Features) Feature weightings learned using state and Q-value data and CAML. Features 1-4 and 6-12 indicate whether a sprite is in the corresponding square-shaped box. Feature 5 is the y-location of the frog. Features 13-17 indicate whether the home locations are empty or filled.

y-position. After that, the positions above and below the frog are assigned the highest weights. Also notice the features to the left of the frog are more highly weighted than the corresponding features on the right. This can likely be attributed to the fact that more sprites are moving from left to right than from right to left.

4.3.3 Irrelevant Features

Comparing the two feature sets, it is clear that using local features results in faster learning. This is because the local feature set contains a relatively small number of states which consists of information required to select a good policy. Contrariwise, the holistic feature set contains a huge amount of information irrelevant to the agent's goals. For example (Figure 4-3), the location of a log at the top of the screen is not very important to the agent controlling the frog at the bottom of the screen. CAML is unable to completely collapse any dimensions from the holistic feature set because each feature becomes important at some point in the frog's trajectory. However, if we add

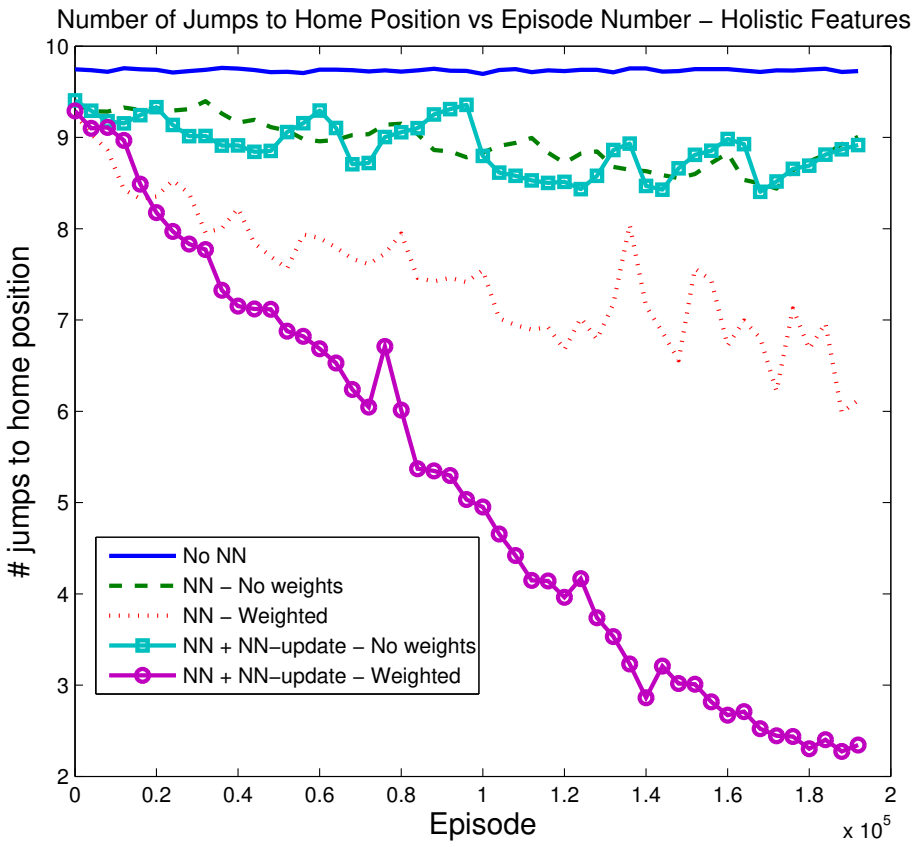

Figure 4-9. (Holistic features) Plot of the minimum number of jumps to home for each episode while learning, convolved with a 5000-length moving average filter, for 5 variants of Q-learning and feature weighting. This measures how close the frog made it to its home before dying in each episode. The low-pass filtering was performed because of extremely large episode-to-episode fluctuations in the data. Each episode corresponds to one frog life.

features truly (or almost) irrelevant to the agent's goals, CAML will ensure these features play a small or no part in action selection and value updates. Figure 4-13 gives a visualization of a modified local feature set. Three additional features were added which indicate whether a sprite is present two jumps below the frog. After 10,000 iterations, we computed feature weights using CAML (shown in Figure 4-8). The "irrelevant" features 10-12 have a low weighting, along with features 1,3,7,9. Figure 4-14 shows the number of games won versus the number of episodes using the nearest neighbor update for

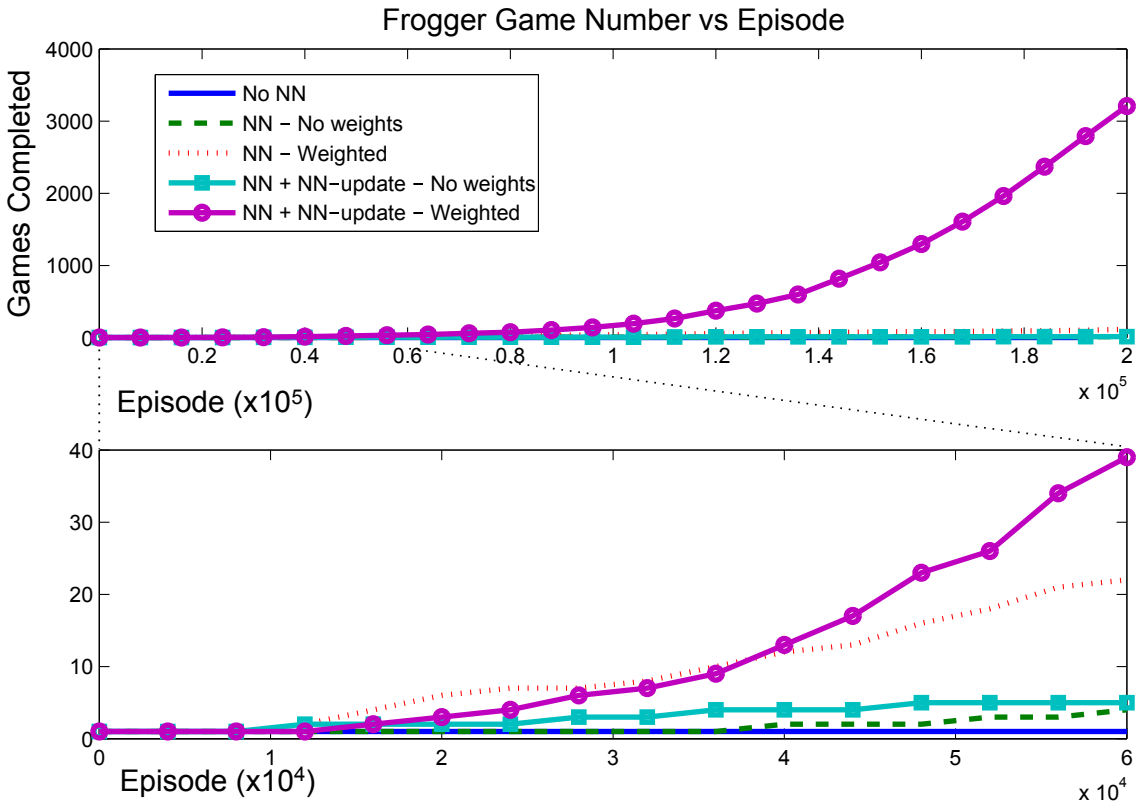

Figure 4-10. (Holistic Features) Cumulative number of games won per episode while learning for five variants of Q-learning and feature-weighting. Each episode corresponds to one frog life. The top subplot shows learning over 200,000 episodes. The bottom subplot shows the same learning curves magnified over the first 60,000 episodes.

weighted and unweighted features. Clearly, the weighted features helped speed up learning.

4.4 Conclusion

We have shown that using nearest neighbor states to bootstrap value function updates in Q-learning can speed up convergence. This approach is particularly useful for game learning tasks, where the state space is very large, but discrete. Standard tabular Q-learning fails in this scenario because states are rarely revisited, and therefore previous experience cannot be exploited. With a Euclidean metric used to find the nearest neighbor, the features of the state must be selected very carefully, since even features that are uncorrelated with the game task will be weighted with equal

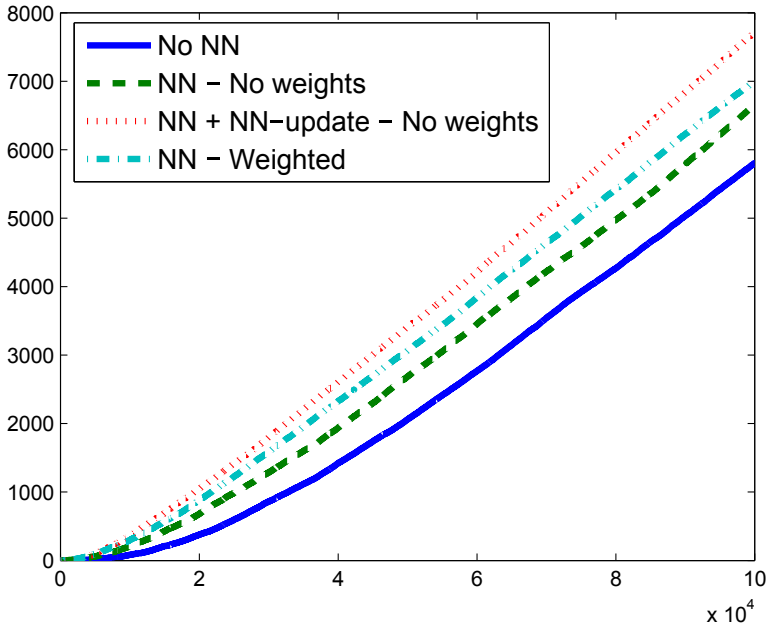

Figure 4-11. (Local Features) Cumulative number of games won per episode while learning for four variants of Q-learning and feature-weighting. Each episode corresponds to one frog life.

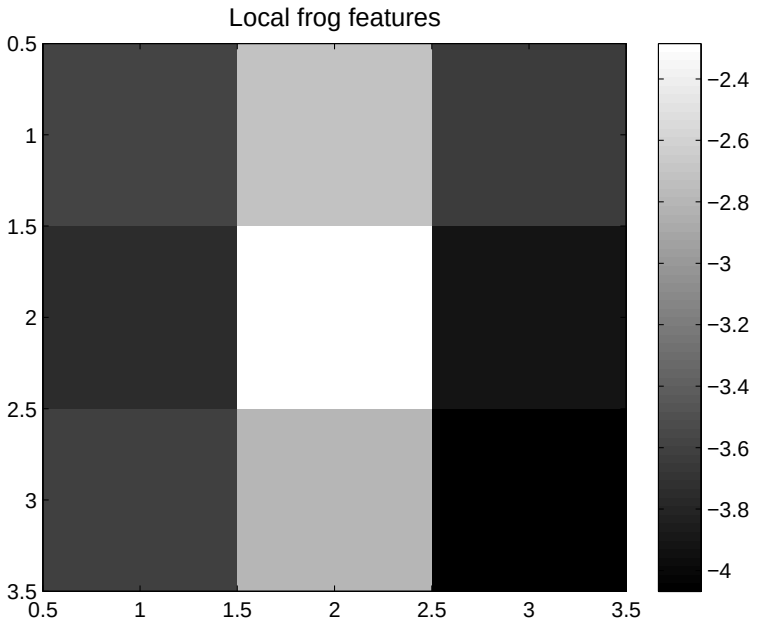

Figure 4-12. Visualization of the weightings for local features.

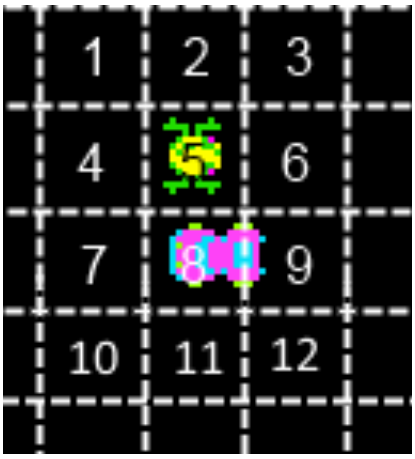

Figure 4-13. Visualization of “local” feature construction with irrelevant features. Features 1-4 and 6-12 indicate whether a sprite is in the corresponding square-shaped box. Feature 5 is the y-location of the frog. Features 10-12 are less relevant in determining the action the agent needs to take. Not pictured here are features 13-17, which indicate whether the home locations are empty or filled.

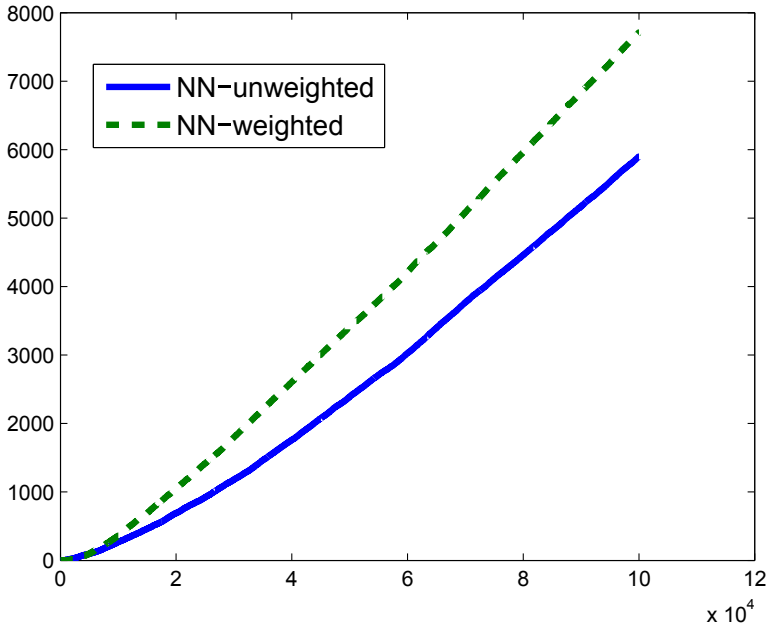

Figure 4-14. (Local + Irrelevant Features) Cumulative number of games won per episode while learning for four variants of Q-learning and feature-weighting. Each episode corresponds to one frog life.

importance. The features can instead be weighted such that the state representation and Q-values are highly dependent. With this weighted metric, neighboring states are also similar with respect to the task (reflected in the Q-value). With this approach, state features can be selected more easily, since irrelevant ones will be de-emphasized by the metric learning. Our method was tested on the classic arcade game Frogger. Two feature sets were used. With a local feature set consisting of the sprites in the immediate vicinity of the frog, nearest neighbor based Q-learning successfully learned how to navigate the frog. When irrelevant features were added to the local feature set, metric learning learned to eliminate or decrease the importance of the irrelevant features when making nearest-neighbor decisions and updates. With a holistic feature set consisting of sprites throughout the entire game map, metric learning greatly improved the learning performance. Our approach for weighting features was used in combination with a simple nearest neighbor approximation. We predict similar success when used with RL methods such as kernel-based RL [58] and RL with Case Based Reasoning (CBR) [59].

CHAPTER 5
DIVERGENCE-TO-GO

The problem of exploration in single state MDPs (i.e. n-armed bandits) has largely been solved [63]. Traditional statistical techniques such as Gittins index [64] or upper confidence bounds [65] are essentially unimprovable [66]. However, exploration of multi-state MDPs remains an open problem currently limited to special cases, such as deterministic MDPs [67]. We propose a reinforcement learning-like framework to guide exploration for the class of MDPs in which it is sensible to assign a metric to its state space.

The most popular exploration strategy is to select a random action with a small probability, epsilon, which can be decreased over time. This epsilon-greedy approach is simple, but is not ideal in practical problems where only a limited amount of interaction with the environment is possible. In such problems, more directed approaches are needed, which very efficiently explore and stop exploring when the task becomes solvable.

A system which intelligently explores the environment requires more information about the progress of learning. In particular, we need to quantify the uncertainty that is present in the values from which decisions are made (e.g. value functions or state transition models). When these quantities are known perfectly, no exploration is needed.

It is a difficult problem to model uncertainty in an RL problem because not only does the amount of uncertainty constantly change, but it is a local and temporal attribute. Knowledge about a state necessarily requires knowledge about future states. However, this problem mirrors the temporal credit assignment problem, which is solved by RL, by using recursion to assign value to states which are part of a sequence. There is a parallel between this management of rewards, and how we can deal with uncertainty. We propose an auxiliary reinforcement learning problem, which uses an information-theoretic quantifier of local uncertainty in place of rewards. There are many potential

choices for which information-theoretic descriptor should be used, entropy being the most obvious. However, when entropy is used to measure uncertainty of an agent in an environment, it is difficult or impossible to separate uncertainty due to stochasticity in the world and uncertainty due to lack of knowledge. This problem can be solved by instead measuring the divergence between the agent's current knowledge of the world and the knowledge that new information brings.

Model-based RL approaches have been shown to be well-suited for the estimation of uncertainty in an RL setting [34], since meaningful quantities can be obtained from probabilities. In model-based RL, the agent interacts with the environment and observes state transitions and estimates a model of the state transition probabilities under the given action. That is, for any point $x \in X$ and $a \in A$, we have a belief distribution which estimates the transition distribution $P(y|x,a)$, where $y \in X$. Each time we observe a transition $x \rightarrow y$, we can update our belief distribution $P(y|x,a)$. Following a belief distribution update, we can compute the divergence $D[P_{new}(y|x,a)||P_{old}(y|x,a)]$ between the two belief distributions, before and after incorporating the new knowledge. This is related to the Bayesian surprise metric, which provides a measure of the information in an observation by comparing the prior and posterior distributions. When a state transition is observed that greatly changes the model, this means that the agent was quite uncertain about the outcome of the action taken in that particular state.

We define the Divergence-to-Go by the expected discounted sum of divergences for the transitions over time,

$$dtg(x,a) = E\left[\sum_{t=0}^{\infty} \gamma^t D(x_t)\right], \tag{5–1}$$

where the expectation is taken over the ensemble of possible paths taken through the state space and the stochasticity is provided by non-deterministic state transitions. Just as the value function in RL characterizes the rewards that will be accumulated during a sequence of actions, the Divergence-to-Go captures the uncertainty of a sequence of actions. In a Markov setting, the sum of divergence values naturally arises because the

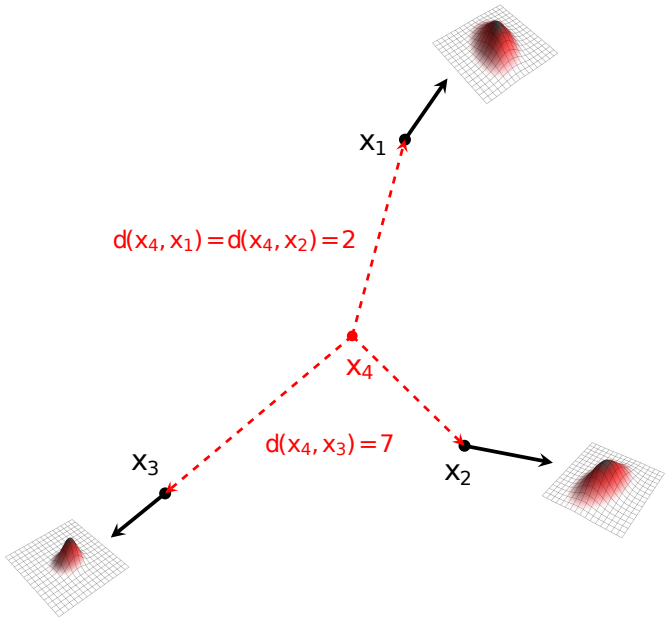

Figure 5-1. Similarity of points. The three black arrows represent sampled transitions from points x_1, x_2, x_3 to some new point in the space. The goal is to model the transition from some new point x_4. The points x_1 and x_2 are closer than x_3 to x_4 with respect to distance metric d.

divergence for the joint transition probability over time becomes a sum of divergences of the marginal transition probabilities. In fact, entropy was used in such a recursive fashion in [68]. We use the divergence between prior and posterior models rather than the entropy because entropy does not differentiate between the randomness that may be inherent in a transition and the randomness that results from an unrefined model.

Unlike a value function, the Divergence-to-Go is not a stationary quantity that we estimate for each state. Instead, as the models improve, the divergence values will decrease to zero. Choosing actions based on the maximum Divergence-to-Go has the effect of maximizing exploration in the sense of visiting the states (or experiencing the state transitions) that most effectively learn the state transition model. Just as with choosing the action with the largest Q-function, the Divergence-to-Go accounts for future uncertainty as well.

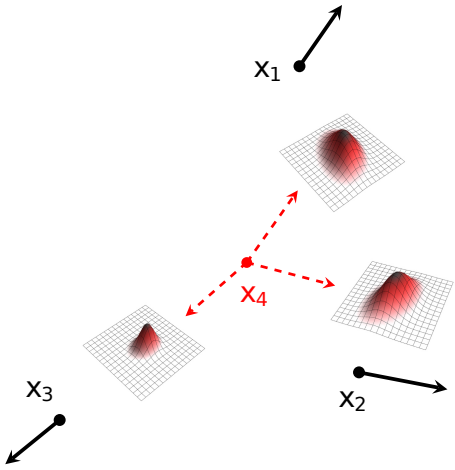

Figure 5-2. Weighting based on similarity of transitions. The points x_1 and x_2 are closer than x_3 to x_4 with respect to distance metric d. In modeling a transition from x_4, we place more weight on the sampled transitions from x_1 and x_2

5.1 Transition Model

Consider Figure 5-1. The black points (x_1, x_2, x_3) and arrows represent state-transitions that have been observed. We wish to estimate the transition model for the red point (x_4). Using the intuitive idea that points close in space have similar transition probability distributions, we choose to weight the contribution of each observation to our model by a similarity measure. We use the similarity function $s'(x, y) = \exp\left\{-\left(\frac{d(x,y)}{\sigma}\right)^2\right\}$, although any suitable similarity function could be used. In Figure 5-2, the contributions made by x_1 and x_2 are much larger than that made by x_3, which is illustrated by the smaller Gaussian bump placed over x_3.

The first assumption we made was that points close in space have similar transition probability distributions. The second assumption we now make is that transition distributions are centered at the initial point of transition. That is, throughout the state space, transition distributions vary smoothly about the initial point of transition. Without this assumption, or some similar weaker one, our model will never be able to generalize. Using this assumption, we center each of our observations around the initial point of the distribution we wish to estimate, weighting them according to the discussion above. With this in mind, suppose we wish to estimate the transition $(x \to y)$ pdf starting from point

x. Then,

$$\hat{p}(y|x) = \sum_i s(x_i, x)\, k_\sigma(y_i - x_i, y - x) \tag{5–2}$$

where

$$s(x_i, x) = \frac{s'(x_i, x)}{\sum_i s'(x_i, x)}. \tag{5–3}$$

because $\sum_i s(x_i, x)$ must sum to 1 so that $\hat{p}(y|x)$ will integrate to 1.

Up to this point we have described a model of a Markov chain consisting of states and transition probabilities (X, P). Recall that a Markov Decision Process (MDP) consists of a 5-tuple (X, A, P, R, γ), where X is the state space, A is the action space, P are the transition probabilities between states given actions, and R is the set of rewards associated with each state-action pair. To fully model this problem, we need to further take into account actions and rewards.

To include actions in this framework, we may assume that similar actions will also lead to similar transition distributions. To model actions $a \in A$, we thus introduce another similarity kernel $s'_a(a_1, a_2)$. Like the state space X, the action space A can be either continuous or discrete. However, unlike the state space, we do not require a discrete action space to accommodate a sensible metric. For action spaces in which a metric can be applied we can define the action similarity kernel as $s'_a(a_1, a_2) = \exp\left\{-\left(\frac{d(a_1,a_2)}{\sigma}\right)^2\right\}$. For action spaces with no metric, i.e., for those with actions which cannot be related to one another, we apply the delta kernel: $s'_a(a_1, a_2) = \delta(a_1 - a_2)$, where δ refers to the Kronecker delta. In this case, each discrete action can be considered as having its own model. We combine the similarity kernels as a product kernel, as we wish to only consider two transitions similar if BOTH the action that caused it and the transition near it are the similar. Thus, the model can then be extended to include actions as:

$$\hat{p}(y|x, a) = \sum_i s^x(x_i, x) s^a(a_i, a)\, k_\sigma(y_i - x_i, y - x) \tag{5–4}$$

To complete the model of the MDP, we include rewards. Rewards can be modeled using some of the same assumptions we used for transitions. That is, similar actions taken at similar states result in similar reward distributions. We model this as:

$$\hat{p}(r|x,a) = \sum_i s_{h_a}(a_i,a)s_{h_x}(x_i,x)k_{\sigma_r}(r_i,r) \tag{5–5}$$

We combine the state and reward models by creating a product kernel $k_{\sigma_y}(y_i - x_i, y - x)k_{\sigma_r}(r_i,r)$:

$$\hat{p}(y,r|x,a) = \sum_i s_{h_a}(a_i,a)s_{h_x}(x_i,x)k_{\sigma_y}(y_i - x_i, y - x)k_{\sigma_r}(r_i,r) \tag{5–6}$$

The above transition model non-parametrically estimates probability density functions using Parzen windows. Thus, ITL-based quantities such as entropy, divergence, and mutual information can be computed using these estimates. We describe some of these quantities below.

5.1.1 Entropy

Renyi's quadratic entropy for the joint state-reward transition distribution can be computed by plugging the pdf estimator into the entropy equation $H_2(X) =$

$-log \int_{-\infty}^{\infty} p^2(x)dx$. Again assuming Gaussian kernel G_σ with kernel size σ:

$$\hat{H}_2(Y, R|x, a) = -\log \int_{-\infty}^{\infty} \hat{p}(y, r|x_t, a_t) \tag{5–7}$$

$$= -\log \int_{-\infty}^{\infty} \left(\sum_i s_{h_a}(a_i, a) s_{h_x}(x_i, x) k_{\sigma_y}(y_i - x_i, y - x) k_{\sigma_r}(r_i, r) \right)^2 \tag{5–8}$$

$$= -\log \int_{-\infty}^{\infty} \sum_i \sum_j s(a_i, a) s(a_j, a) s(x_i, x) s(x_j, x) \times \tag{5–9}$$

$$k_{\sigma_y}(y_i - x_i, y - x) k_{\sigma_y}(y_j - x_j, y - x) k_{\sigma_r}(r_i, r) k_{\sigma_r}(r_j, r) \tag{5–10}$$

$$= -\log \sum_i \sum_j s(a_i, a) s(a_j, a) s(x_i, x) s(x_j, x) \times \tag{5–11}$$

$$\int_{-\infty}^{\infty} k_{\sigma_y}(y_i - x_i, y - x) k_{\sigma_y}(y_j - x_j, y - x) k_{\sigma_r}(r_i, r) k_{\sigma_r}(r_j, r) \tag{5–12}$$

$$= -\log \sum_i \sum_j s(a_i, a) s(a_j, a) s(x_i, x) s(x_j, x) k_{\sigma_y}(y_i - x_i, y_j - x_j) k_{\sigma_r}(r_i, r_j) \tag{5–13}$$

Furthermore, the quadratic information potential $\hat{V}_2(X)$ is estimated as the argument of the log so that

$$\hat{V}_2(Y, R|x, a) = \sum_i \sum_j s_{h_a}(a_i, a) s_{h_a}(a_j, a) s_{h_x}(x_i, x) s_{h_x}(x_j, x) k_{\sigma_y}(y_i - x_i, y_j - x_j) k_{\sigma_r}(r_i, r_j) \tag{5–14}$$

We write this out in full to be explicit in the computation involved. In the interest of conciseness, we rewrite Equation 5–14 by joining the similarity kernels s. In this context, x will be taken to mean a state-action vector. Furthermore, we omit the reward kernel unless required.

$$\hat{V}_2(Y|x, a) = \sum_i^N \sum_j^N s(x_i, x) s(x_j, x) k(y_i - x_i, y_j - x_j) \tag{5–15}$$

Renyi's quadratic cross-entropy is defined as $H_2(p_1, p_2) = -\log \int_{-\infty}^{\infty} p_1(z) p_2(z) dz$. Suppose $\{x_i^1 \to y_i^1, r_i^1\}_{i=1}^N$ and $\{x_i^2 \to y_i^2, r_i^2\}_{i=1}^M$ are sampled transitions, associated with

p_1 and p_2. Following a similar procedure as above, Renyi's quadratic cross-entropy can be estimated as:

$$\hat{H}_2(p_1, p_2) = -\log \sum_i^N \sum_j^M s_{h_a}(a_i^1, a) s_{h_a}(a_j^2, a) s_{h_x}(x_i^1, x) s_{h_x}(x_j^2, x) k_{\sigma_y}(y_i^1 - x_i^1, y_j^2 - x_j^2) k_{\sigma_r}(r_i^1, r_j^2). \tag{5–16}$$

As above, for the sake of economy, we may shorten this as

$$\hat{H}_2(p_1, p_2) = -\log \sum_i^N \sum_j^M s(x_i^1, x) s(x_j^2, x) k(y_i^1 - x_i^1, y_j^2 - x_j^2). \tag{5–17}$$

5.1.2 Divergence

Given an initial state-action pair, the Euclidean and Cauchy-Schwarz divergences can be computed between two state-reward transition distributions. Denoting the two transition distributions as p and q, Euclidean and CS divergence is computed as:

$$D_{euc}(p, q) = \int (p(x) - q(x))^2 \, dx = \int p^2(x) dx - 2 \int p(x) q(x) dx + \int q^2(x) dx \tag{5–18}$$

$$D_{cs}(p, q) = -\log \frac{\left(\int p(x) q(x)\right)^2}{\int p^2(x) dx \int q^2(x) dx}. \tag{5–19}$$

It can be seen that both equations consist of solely information potential and cross-information potential terms

$$\hat{D}_{euc} = \hat{V}_f + \hat{V}_g - 2\hat{V}_c \tag{5–20}$$

$$\hat{D}_{cs} = \log \frac{\hat{V}_f \hat{V}_g}{\hat{V}_c^2}, \tag{5–21}$$

which can be estimated as

$$\hat{V}_f = \sum_{i=1}^{N}\sum_{j=1}^{N} s_f(x_i, x)s_f(x_j, x)\, G_{\sigma\sqrt{2}}(y_i - x_i, y_j - x_j) \tag{5–22}$$

$$\hat{V}_g = \sum_{i=1}^{M}\sum_{j=1}^{M} s_g(x_i, x)s_g(x_j, x)\, G_{\sigma\sqrt{2}}(y_i - x_i, y_j - x_j) \tag{5–23}$$

$$\hat{V}_c = \sum_{i=1}^{N}\sum_{j=1}^{M} s_f(x_i, x)s_g(x_j, x)\, G_{\sigma\sqrt{2}}(y_i - x_i, y_j - x_j). \tag{5–24}$$

Each of the potentials (V) are clearly quadratic forms, and can be calculated using kernel matrices and similarity vectors. That is, if $K_{ij} = G_{\sigma\sqrt{2}}(y_i - x_i, y_j - x_j)$, and $\mathbf{s}_i = s(x_i, x)$, then $\hat{V} = \mathbf{s}^T K \mathbf{s}$.

5.1.3 Mutual Information

A number of mutual information types have been defined using the information-theoretic learning framework. Two, based on the Euclidean and Cauchy-Schwarz divergences are defined as the divergence between joint and marginal distributions. Again, suppose $\{x_i^1 \to y_i^1, r_i^1\}_{i=1}^N$ and $\{x_i^2 \to y_i^2, r_i^2\}_{i=1}^N$ are sampled transitions, where we assume each pair of the transitions i occur jointly. Then the joint distribution can be computed by concatenating the samples so that $\mathbf{x}_i = \left[x_i^1, x_i^2\right]^T$, $\mathbf{y}_i = \left[y_i^1, y_i^2\right]^T$, and $\mathbf{r}_i = \left[r_i^1, r_i^2\right]^T$. Thus, $\{\mathbf{x}_i \to \mathbf{y}_i, \mathbf{r}_i\}_{i=1}^N$. Then, the divergence can be computed between the product of the marginal and joint distributions as discussed above.

5.2 Divergence-to-Go

Since divergence-to-go is a dynamic programming framework, we must solve it with analogous methods to those in RL. In continuous state-action spaces, we must map state-action pairs to dtg values. We use the Kernel Temporal Difference (KTD) algorithm [44], which can be considered as a kernelized version of Q-learning. Let $\bar{x} := (x, a)$ be the state-action pair. We compute divergence-to-go with KTD using the product kernel

over $\bar{x}$,

$$\delta_t = D + \gamma \max_a \{dtg_{t+1}(\bar{x}')\} - dtg_t(\bar{x}) \tag{5–25}$$

$$dtg_t(\bar{x}) = D_0 + \alpha \sum_{j=1}^{t} \delta_j k(\bar{x}, \bar{x}_j), \tag{5–26}$$

where D is a any divergence measure, α is the learning rate, γ is the discount factor, and D_0 is some initial value of dtg.

Divergence is computed according to Equations (5–20) and (5–21). For each divergence computation, we use only the N nearest neighbors to $\bar{x}$. Suppose the N nearest neighbors are the samples $(\bar{x}_1, \ldots, \bar{x}_{N/2}, \ldots, \bar{x}_N)$ ordered in time from oldest to most recent. The divergence is then computed between the estimated distributions that are generated by the samples $(\bar{x}_1, \ldots, \bar{x}_{N/2})$ and $(\bar{x}_{N/2+1}, \ldots, \bar{x}_N)$. Using only the nearest neighbors serves three purposes. First, it allows the computation of divergence between two distributions of equal number of samples. This allows a fair comparison between distributions computed when there a small number of samples in the beginning of learning, and when there are large number of samples when learning is almost complete. Second, it speeds up computation, as the divergence computation scales at $O(N^2)$. Finally, the appropriate kernel size for Parzen density estimation is dependent on the number of samples used in the estimation. Using only the N nearest neighbors allows setting of kernel size according to Silverman's rule. For univariate distributions, Silverman's rule suggests the optimal kernel size is:

$$1.06 \min \{\sigma, R/1.34\} N^{-1/5}, \tag{5–27}$$

where N is the number of samples, σ is the standard deviation of the samples and R is the interquartile range. For multivariate distributions of dimension D, Silverman's rule suggests the optimal kernel size is somewhere in the interval:

$$\left[1.06 \min \{\sigma, R/1.34\} N^{-1/5}, 1.06 \min \{\sigma, R/1.34\} N^{-1/(5D)}\right], \tag{5–28}$$

where N is the number of samples, σ is the standard deviation of the samples, R is the interquartile range, and D is dimensionality of the samples.

The discount factor plays an important role in the divergence-to-go framework. Often RL problems are episodic in nature. The expected sum of rewards is then computed over each episode. Without episodes, the expected sum of rewards, or in the case of dtg, divergences, over an infinite number of time steps itself may approach infinity. Clearly, estimating such a quantity is impractical. Dtg does not require and, in most cases, does not use an episodic framework. The discount factor then serves as a approximate measure of the number of steps into the future over which we compute the sum of divergences. We use the thumb rule that $\gamma^n < 0.1$ is small enough that future divergences contribute a negligible amount to the total sum. For each problem, we set the discount factor based on the number of steps we expect the agent to require to move from any point in the state space to any other point in the state space. Using the thumb rule we solve for $\gamma^N = 0.1$:

$$\gamma = 10^{-\frac{1}{N}} \tag{5–29}$$

The initial DTG value D_0 is the value assigned to each state-action combination prior to any update. Assigning a small value to D_0 results in the dtg policy exploring small areas of the state space until the estimated divergence-to-go for the actions in that area are below D_0. Conversely, assigning a large value to D_0 results in a dtg policy which explores the entire space before focusing on areas of with high divergence (i.e., areas with rapidly changing distributions).

Let us consider a principled method to assign a value to initial dtg D_0 using the idea of maximum divergence D_{max}. D_{max} is the maximum divergence expected to be computed while learning the transition model. Since divergence decreases from D_{max} to approach zero as the model is being learned, it makes sense to set D_0 as the maximum possible dtg. Recall dtg is defined as $E\left[\sum_{i=0}^{\infty} \gamma^i D_i\right]$. Replacing D_i with D_{max}, we get

$\sum_{i=0}^{\infty} \gamma^i D_{max}$ which converges to $\frac{D_{max}}{1-\gamma}$ Thus, we set:

$$D_0 = \frac{D_{max}}{1-\gamma} \tag{5–30}$$

For another point of view, consider the TD error Equation (5–25) and suppose D_{max} is observed. Assuming the initial value D_0 is assigned to both $\max_a dtg_{t+1}$ and $dtg_t(\bar{x})$, we expect the TD-error to be zero. That is, after observing the maximum divergence, we do not wish the dtg estimate for $dtg_t(\bar{x})$ to be decreased from its initial value. Substituting D_{max} and D_0 into Equation (5–25), we arrive at $D_{max} + \gamma D_0 - D_0 = 0$. Solving for D_0, we again arrive at Equation (5–30).

In practice, depending on the constraints of the problem, it can be difficult or impossible to compute D_{max}. Fortunately, maximum divergence can be usually be estimated empirically after a single rough sweep through the state space. Finally, if desired, D_0 can be adjusted by a multiplier κ using

$$D_0' = \kappa D_0 = \frac{\kappa D_{max}}{1-\gamma}. \tag{5–31}$$

Large values of $\kappa > 1$ weight initial exploration toward evenly exploring the space. Small values of $\kappa < 1$ weight initial exploration toward exploring anomalies.

5.3 Augmented Divergence-to-Go

When the agent is making a decision, divergence-to-go as discussed above asks the question: “Given the state of the world, which action should be selected so as to maximize the expected sum of future divergence in the model?” This approach is appropriate for keeping track of which states-actions have been visited, and for revisiting states-actions proportional to the divergence they produced. However, this approach is not appropriate for learning a policy to be used in similar environments. That is, as training time $\to \infty$, $\sum_{x \in X, a \in A} dtg(x, a) \to 0$.

When the goal is to learn such a policy, we must modify our approach. The new question we must ask is “Given the state of the world, *and* the condition of our model,

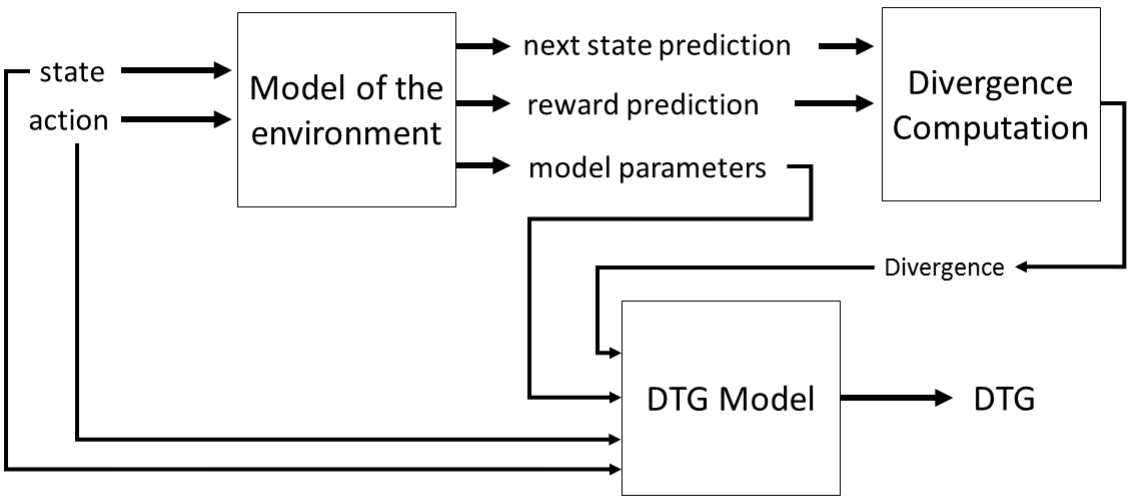

Figure 5-3. Divergence-to-go with augmented state vector block diagram. This demonstrates the the computation/update performed at each iteration. The transition model block takes a state and an action as an input to produce a joint probability density function of the next state and reward. Divergence is computed from two model pdfs produced using different sampled transitions. The DTG block updates the augmented state vector of states, actions, and transition model parameters with the divergence computed by the divergence computation block.

which action should be selected so as to maximize the expected sum of future divergence in the model?" To answer this question, we must specify how to quantify the condition of the model.

The condition of the model can be quantified either by the parameters of the model or the output the model produces. Two approaches seem apparent: a global approach and a local approach. The global approach summarizes how the transition model acts globally, i.e. for every state and action. A simple scheme to implement this approach is to use the model's (global) parameter vector. On the other hand, the local approach summarizes how the model acts at a certain state or for a state-action pair. A straightforward scheme to implement this approach is to use the transition model

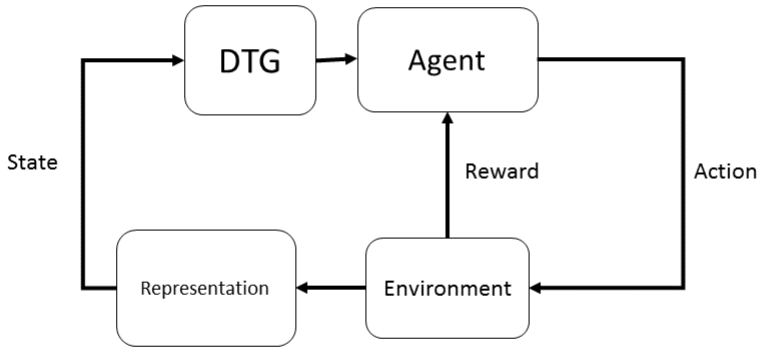

Figure 5-4. Block diagram of perception-action cycle with divergence-to-go. Agent acts within the environment which in turn produces a new state.

parameters which affect how the model works at a specific state. Another scheme is to summarize the local ouput of the model by paramaterizing the transitions from a certain state or state-action pair.

The transition model described in 5.1 does not have easily accesible parameters which can be used to summarize the model. Furthermore, it is difficult to summarize the global output of the model. Thus, we use the local approach and summarize transitions using ITL quantities. Given a state-action pair, we represent a transition using the transition probability density function's mean and entropy. The mean specifies the pdf's location, while the entropy specifies the pdf's shape.

Thus, we augment the state-action input vector with the transition pdf mean and entropy. To accomplish this, we introduce the mean kernel $k_m(\cdot,\cdot)$ and the entropy kernel $k_h(\cdot,\cdot)$. We select the Gaussian kernel for both. Then for divergence-to-go, we define the product kernel $k((x, m, h, a), (x', m', h', a')) = k_x(x, x')k_m(m, m')k_h(h, h')k_a(a, a')$.

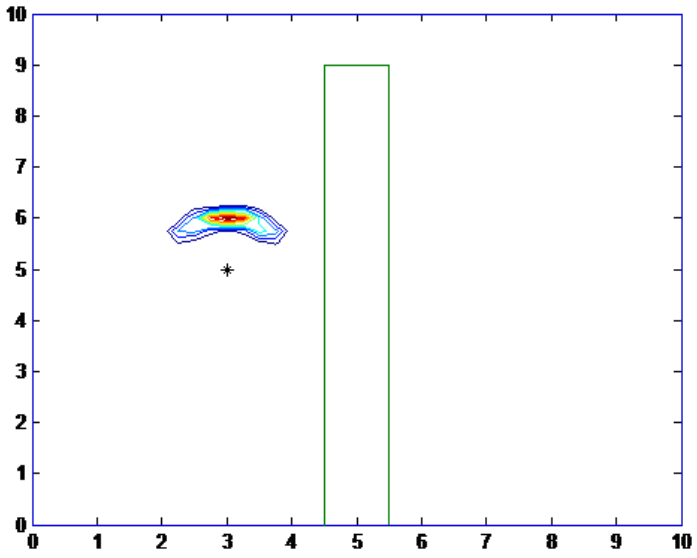

Figure 5-5. Contours of a learned transition pdf in response to the "up" action in a maze environment. For each action, the agent moves one unit at an angle chosen probabalistically according to a Gaussian distribution with standard deviation $\pi/6$, and mean the direction chosen.

This kernel is positive definite and similar to the discussion on KTD in Section 3.1, $k((x, a, h, a), (\cdot, \cdot, \cdot, \cdot) = \varphi_x(x) \otimes \varphi_m(m) \otimes \varphi_h(h) \otimes \varphi_a(a)$ which is defined in the feature space. The dtg update is then

$$dtg(x_n, a, m, h) = \eta \sum_j^n \delta_j k((x, a, m, e), (x_j, a_j, m_j, h_j)) \tag{5–32}$$

where

$$\delta_i = D_i + \gamma \max_a dtg(x_{n+1}, m_{n+1}, e_{n+1}, a) - dtg(x_n, a_n) \tag{5–33}$$

5.4 Results: Exploration in a Maze

5.4.1 Two-Dimensional Maze

We now present results for a simple two-dimensional maze. For this simulation, the agent starts at the point (1,1). Similarly to the agent in Figure 5-15, at each time step, it must select an action $a \in [0, 2\pi)$. Each of these actions moves the agent one unit in a direction $a + \theta$ where θ is drawn from $\mathcal{N}(0, (\pi/6)^2)$. In order to learn a model of the

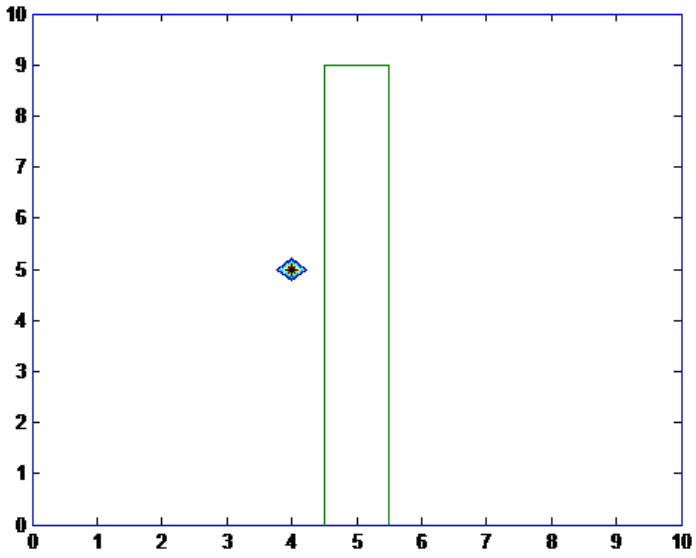

Figure 5-6. Contours of a learned pdf in response to the "right" action in a maze environment next to a wall. For each action, the agent moves one unit at an angle chosen probabalistically according to a Gaussian distribution with standard deviation $\pi/6$, and mean the direction chosen. A transition which would result in the agent moving through a wall results in the agent remaining in place. The contours are thus centered at the agent.

entire space, and since the dynamics of the space do not change throughout the maze, divergence-to-go will attempt to explore the entire maze in a relatively even fashion. In Figure 5-5 we show a contour plot for a transition pdf from [3,5] and action up. For the transition model, we used a similarity kernel size of 0.5 and a probability kernel size of 0.15. Similarly, in Figure 5-6, we show the pdf learned for selecting the action "right," while at position [4,5]. This pdf is centered at the initial point because attempting to move right at that point results in the agent not moving.

After exploring the maze using divergence-to-go, we assigned a reward of one to the sample closest to (10,0) and ran policy iteration using the learned probability transition model. Since we are working with samples, we normalize the transition probabilities so that $\sum_i P(x_j, a, x_i) = 1$ for each x_j in the sample space and each $a \in A$.

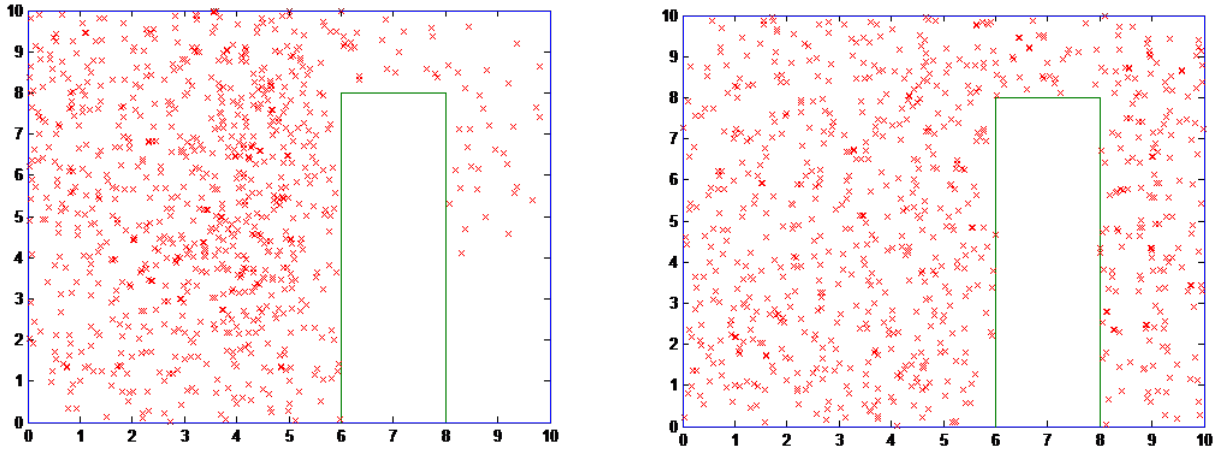

Figure 5-7. (left) Points visited while using random exploration policy. (right) Points visited while using divergence-to-go based exploration policy.

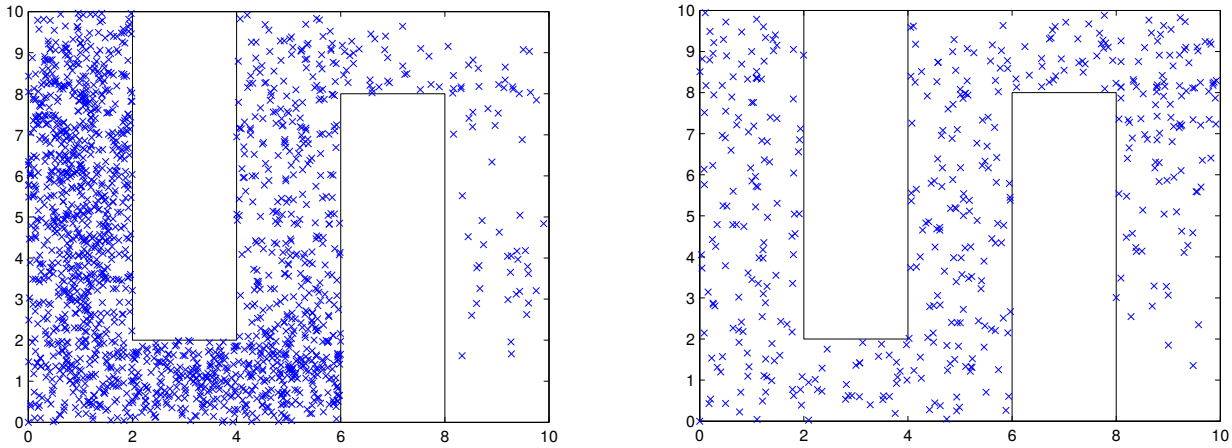

Figure 5-8. With dtg, the agent quickly spans the maze very uniformly in comparison to random exploration.

That is, the approximating assumption that samples can transition to only each of the other samples is made.

Using this assumption, we create a transition probability matrix P for angles $\{0°, 5°, 10°, \ldots, 355°\}$. Let $P^{a}_{xx'}$ be the transition probability from x to x' given action a. The value function estimate for each state is then updated iteratively according to

$$V(x) \leftarrow \sum_{x'} P^{\pi(x)}_{xx'}[R + \gamma V(x')]. \tag{5–34}$$

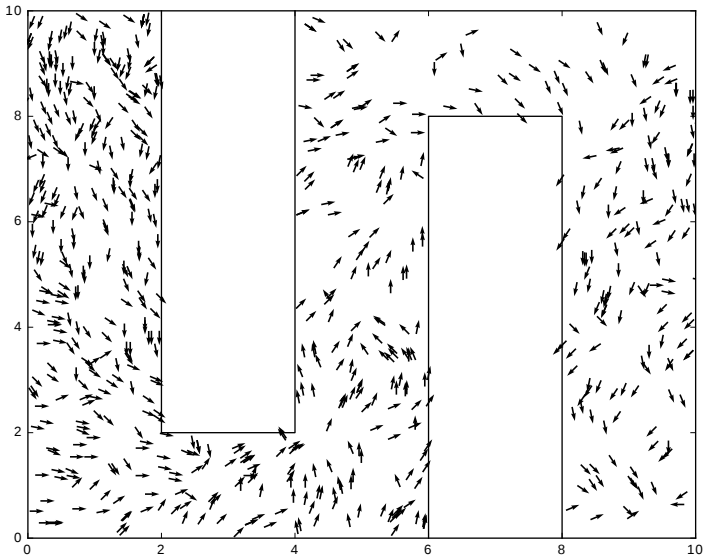

Figure 5-9. Actions learned for each point visited using policy iteration and learned transition model

Then, the policy is updated according to

$$\pi(x) \leftarrow \underset{a}{\operatorname{argmax}} \sum_{x'} P^{a}_{xx'} \left(R(x, a) + \gamma V(x') \right) \tag{5–35}$$

This process is repeated until convergence. Figure 5-9 shows the learned optimal action for each sample point after the goal reward was propagated through all the samples in the space.

The plot shown in Figure 5-7 shows every point visited in a typical simulation where random actions were selected every time step (1000 time steps). There are large “clumps” of points, and the whole space was not even explored. Similarly, 5-7 also shows every point visited while selecting the maximum divergence-to-go actions. It can be seen points are more evenly spaced across the entire space.

Figure 5-8 shows each point visited in the maze for two representative simulations of random exploration and dtg-based exploration. Both simulations were run until the agent reached the goal position. The random exploration policy slowly reaches the goal,

making a diffusion-like state pattern. The dtg policy swiftly reaches the goal by exploring the maze as quickly as possible.

Finally, we compared divergence-to-go exploration with random exploration. We ran 100 Monte Carlo trials for each type of exploration and recorded the number of steps n_{steps} it took to reach within $\sqrt{2}$ of the bottom right corner (10,0). Table 5-1 gives the results. Divergence-to-go is clearly better in every statistic. In the mean, divergence-to-go is able to reach the goal by a factor of about five times faster. As one would expect, random exploration results in a huge variance between trials, while divergence-to-go is relatively consistent.

Table 5-1. Divergence-to-go vs random exploration in a 2-d maze

n_{steps}	mean	max	min	std
DTG	487.77	1651	145	311.47
Random	2486.1	8887	327	2019.9

5.4.2 Three-Dimensional Maze

Furthermore, we created a 3-dimensional maze for the agent to explore. The agent starts at the coordinates (1,9,1). This time, the agent has six actions to select from: north, east, south, west, up, and down. Each action moves the agent one unit in a direction determined probabilistically by two Gaussian distributions of standard deviation $\pi/6$ centered at the action's direction. That is, each action can cause the agent to land somewhere on a sphere of radius one centered at the agent. We used a similarity kernel size of 0.5 and a probability kernel size of 0.1.

We allowed the agent to explore the maze, using both methods, until it discovered the goal of being within $\sqrt{3}$ of the goal at coordinates (1,9,1). Figure 5-10 shows every point visited during a typical simulation using random exploration from a front-facing view of the maze and a top-down view. Figure 5-11 shows the same data for a typical divergence-to-go simulation. The difference between the two is immediately apparent.

Again, we ran 100 monte carlo trials and recorded the number of steps n_{steps} it took to reach within $\sqrt{3}$ of the bottom right corner (1,9,1). Table 5-2 gives the results.

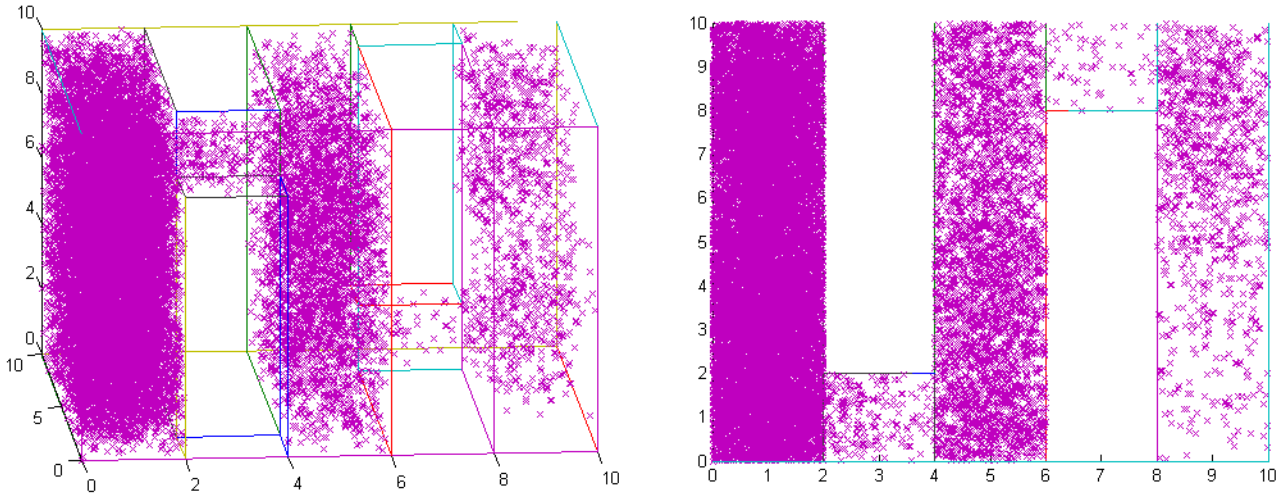

Figure 5-10. Points visited in 3-d maze while following random exploration policy — (left) front view of maze (right) top view of maze

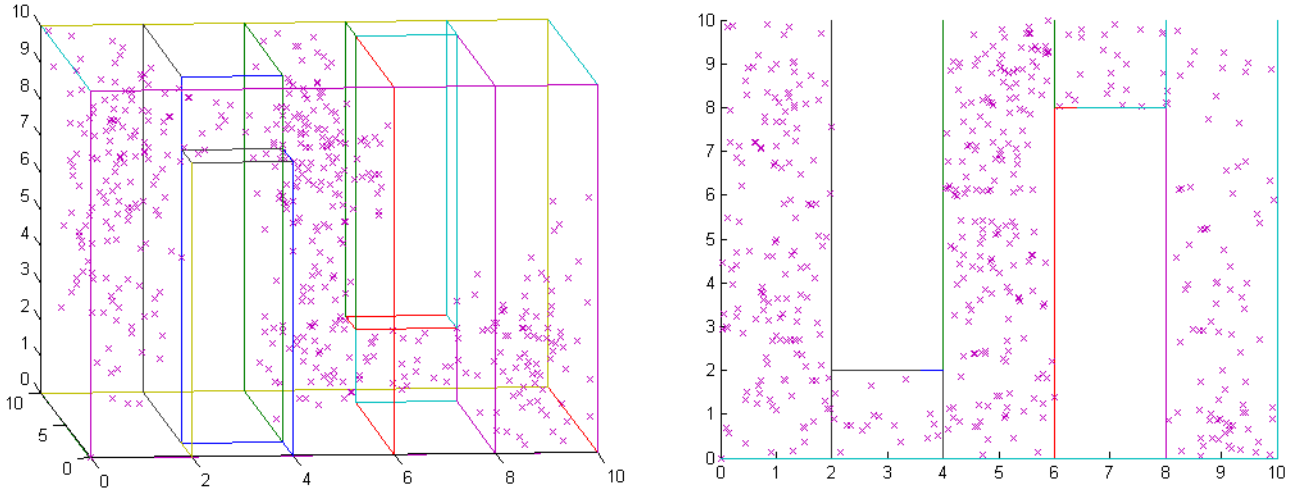

Figure 5-11. Points visited in 3-d maze while following divergence-to-go based exploration policy — (left) front view of maze (right) top view of maze

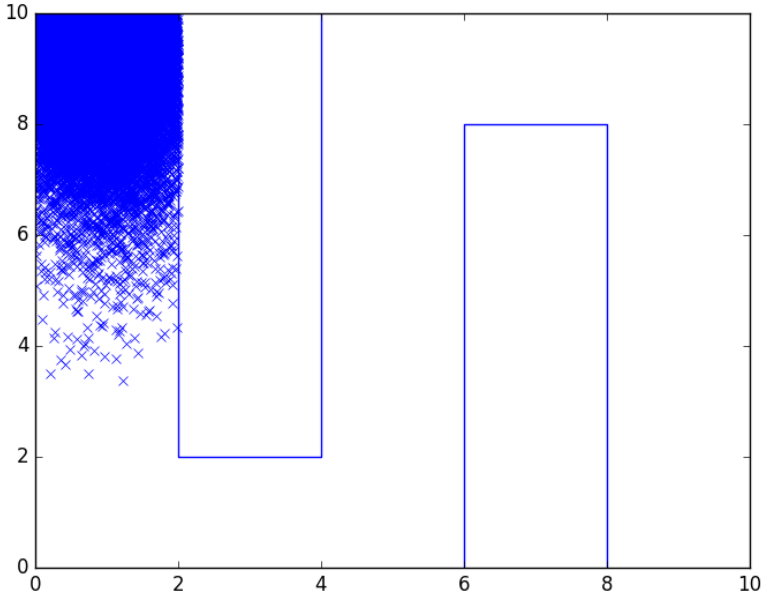

Figure 5-12. States visited for the 2-d maze experiment with resets while following a random policy. Each time the agent attempts to move into the wall, it is reset to its initial position.

Once again, divergence-to-go is clearly better in every statistic. This time, in the mean, divergence-to-go is able to reach the goal by a factor of about 3.5 times faster.

Table 5-2. Divergence-to-go vs random exploration in a 3-d maze

n_{steps}	mean	max	min	std
DTG	2711.8	7912	608	1381.2
Random	9845.1	54535	880	9439.8

5.4.3 Maze with Resets

To test the ability of divergence-to-go to learn a policy which maximizes exploration for new transition models under similar environments, we introduce the maze with resets. For this experiment we use the same 2-d maze, but now the agent is reset to the starting position whenever the transition intersects with a wall. The action space $A = [-\pi, \pi]$. Each action moves the agent 0.5 units in a direction determined probabilistically by a Gaussian with standard deviation $\pi/6$ centered at the respective action direction.

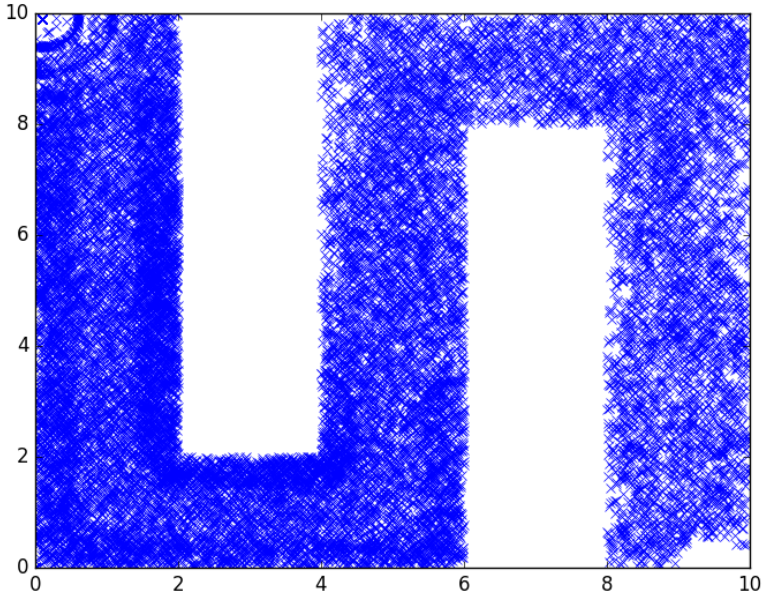

Figure 5-13. States visited for the 2-d maze experiment with resets while following a divergence-to-go policy. Each time the agent attempts to move into the wall, it is reset to its initial position.

For this experiment a policy was learned using the dtg framework. After a policy was learned for the maze reset environment and transition model, the transition model was reset, and the agent was made to follow the dtg policy.

Divergence-to-go should learn a policy which will improve the model the most in the smallest number of steps. After a few initial iterations, choosing actions which move the agent into any wall causes the agent to be reset to a location in the state space that the model has already learned well. Thus, divergence-to-go should learn a policy which avoids the walls until most of the space has been thoroughly explored.

Figures 5-12 and 5-13 show the locations in the maze explored after 100,000 iterations while following the random and dtg policies respectively. The random policy is only able to explore approximately 1/8 of the maze. The DTG policy eventually learns to explore the entire maze by learning where the walls are and avoiding them.

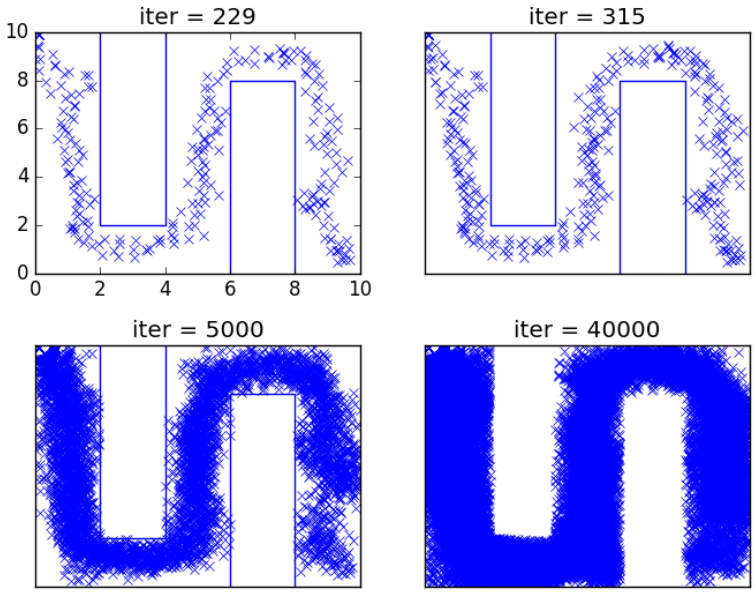

Figure 5-14. Plot of each state visited at four different iterations in the "reset" maze experiment while following a learned dtg policy. Each time the agent attempts to move into the wall, it is reset to its initial position. Iterations 229 and 315 correspond to the first two times the agent completes the maze.

Figure 5-14 shows, after resetting the transition model to a clean slate, exploration following the newly learned DTG policy after 229, 315, 5000, and 40000 iterations. Iterations 229 and 315 are the first two times the agent reaches the end of the maze. The DTG policy improves the new transition model by avoiding the walls and exploring the entire maze.

5.4.4 Open Maze

To test the ability of divergence-to-go to more thoroughly explore areas of the state space with a changing distribution, we introduce the open maze experiment. For this experiment we use the same 2-d maze environment, but for the area $5 < x, y < 7$, each action distribution is rotated by π radians. For this experiment, there are no inner walls. Again, $A = [0, 2\pi)$ and each of these actions moves the agent one unit in a direction $a + \theta$ where θ is drawn from $\mathcal{N}(0, (\pi/6)^2)$. A plot of the learned transition distributions for the actions $\{0, \pi/4, \pi/2, 3\pi/4, \pi, 5\pi/4, 3\pi/2, 7\pi/4\}$ are shown in Figure 5-16. Actions

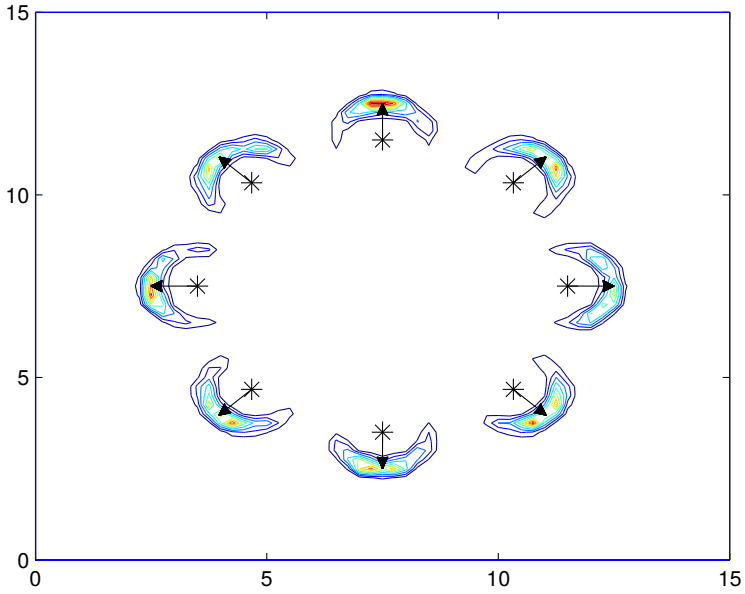

Figure 5-15. Contour plot of model-learned transitions pdfs in an open maze for eight actions and a continuous action set $A = [0, 2\pi)$. For each action, the agent moves one unit at an angle chosen probabalistically according to a Gaussian distribution with standard deviation $\pi/6$, and mean the direction chosen.

causing the agent to move into the wall at the edges of the maze were assigned zero divergence.

To explore this maze, the agent followed a divergence-to-go exploration policy for 10 Monte Carlo trials of 20000 iterations. We then performed kernel density estimation using every point visited in the space, averaging over the 10 Monte Carlo trials. Figure 5-16 shows a heat map of the result. Red indicates the areas visited most frequently, while blue indicates the areas visited least frequently. The dashed line indicates the area where the action distribution changes.

The points most visited were near the border at which the action-conditioned distributions change. The interior of this area was also highly explored as the agent only had a small space to fully learn the new distributions. The outer borders of the maze were the least visited due to zero divergence being assigned to the actions causing

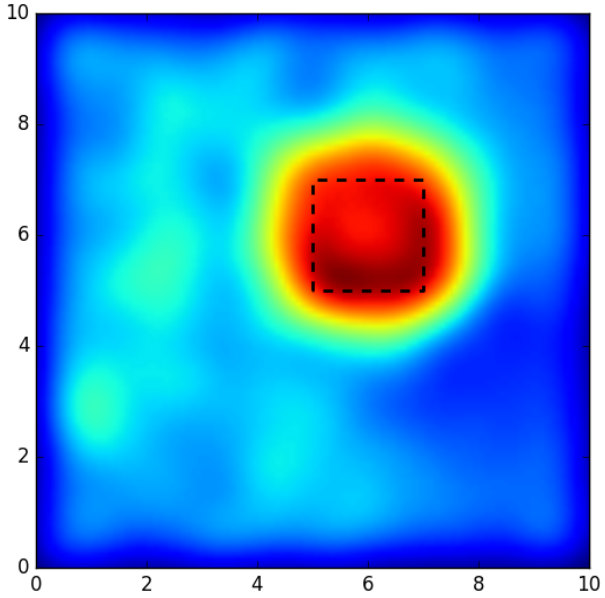

Figure 5-16. Heat map of states visited for the open 2-d maze experiment. The states inside the black dashed box have a different transition distribution from the rest of the states. Red corresponds to heavily visited states, while dark blue corresponds to lightly visited states.

the agent to attempt to move outside the boundaries of the maze. The top left area of the maze was visited more than the bottom right because it was the agent's starting location.

5.4.5 Mountain Car

Recall the mountain car problem described in described in 3.2.1 and Figure 3-1. The car begins at the bottom of the hill (position 0.5) and the agent attempts to drive it to the goal position at the top right. However, the car lacks sufficient power to drive directly to the goal, so it first must climb the left hill and allow the combination of gravity and the car's power to drive it up the right hill. The mountain car state space consists of two variables: the position p and the velocity v of the mountain car.

Divergence-to-go was applied to the mountain car problem. The state space and action spaces were normalized so that the range of each dimension is one. The similarity kernels were set to .05, and the state kernel size was set to .001. Furthermore,

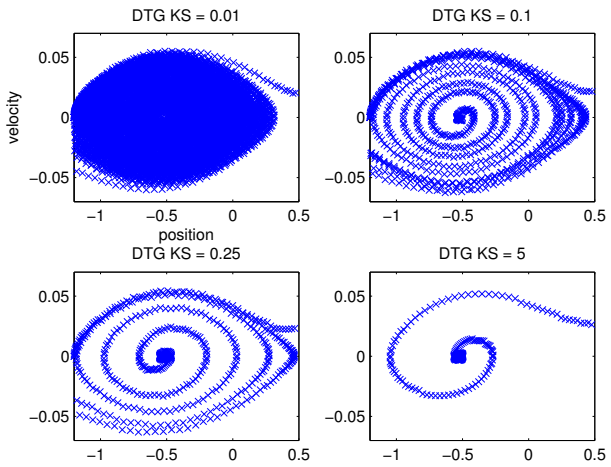

Figure 5-17. Mountain car exploration. The states visited during training for four kernel sizes.

a divergence-to-go learning rate α of 0.75 and a discount factor γ of 0.9 were used. The action space $\mathcal{A}_{cont}$ was used. Figure 5-17 shows each point visited in the trajectory of the mountain car state space from at rest at the bottom of the hill to the goal state for various divergence-to-go kernel sizes. Smaller kernel sizes lead to a better model as the state/action space is more thoroughly sampled while larger sizes cause the space to be explored more quickly.

Table 5-3 shows the number of steps until exploration by divergence-to-go reaches the goal state for the four kernel sizes shown in Figure 5-17. Furthermore, the average number of steps required to reach the goal over 1000 Monte Carlo trials are shown for both a random policy and (tabular) SARSA [69] over a discretized state space. The following parameters were used for SARSA: a learning rate α of 0.7, an eligibility trace parameter λ of 0.3, an ϵ-greedy exploration parameter ϵ of 0.1, and a discount parameter γ of 1. Furthermore, the action space $\mathcal{A}_{disc}$ was used. Table 5-3 also shows the number of steps required to reach the goal after the policy learned by divergence-to-go is improved using 250 episodes of Monte Carlo tree search (MCTS). The results show that pure exploration is able to reach the goal more quickly than a traditional reinforcement learning method on a single-episode basis.

Table 5-3. Number of steps from cart at rest at bottom of hill to goal for various dtg kernel sizes

dtg ks	0.01	0.1	0.25	5	rand	SARSA
#steps	9657	857	511	231	41994	3792
#steps (MCTS)	1026	488	235	151	-	-

5.4.6 Coil-100 Environment

Until now, the experiments of consisted of two or three dimensional state space. We now discuss applying divergence-to-go to higher-dimensional environments. The COIL-100 dataset [70] consists of 100 videos of different objects. Each video is 72 frames long and were generated by placing each object on a turntable and taking a picture every 5 degrees. The pictures are 128x128 pixels RGB for a dimensionality of 49152. For our experiments, we rescaled the images to 32x32 pixels for a dimensionality of 3072.

Typically, when testing classifier accuracy with this dataset, the frames corresponding to $0°, 90°, 180°$, and $270°$ are used to train a classifier and the rest are used for validation and test. However, many experimenters have used the entire dataset to first learn a representation before training the classifier. For example, Chalasani and Principe [71] [72] trained a convolutional deep predictive coding network (DPCN) using the entire dataset, but then trained a classifier using the DPCN representation using only $0°, 90°, 180°$, and $270°$.

We, however are not interested in the images themselves, but in modeling the transitions from any object angle to any other arbitrary angle. These transitions will differ for each object. We thus follow another protocol. The following experiment consists of three parts.

- First, we learn an unsupervised lower dimensional representation for the data. This both facilitates faster learning and a provides a representation with many ambiguous overlapping samples. This ambiguity provides a more difficult environment for learning an accurate model.

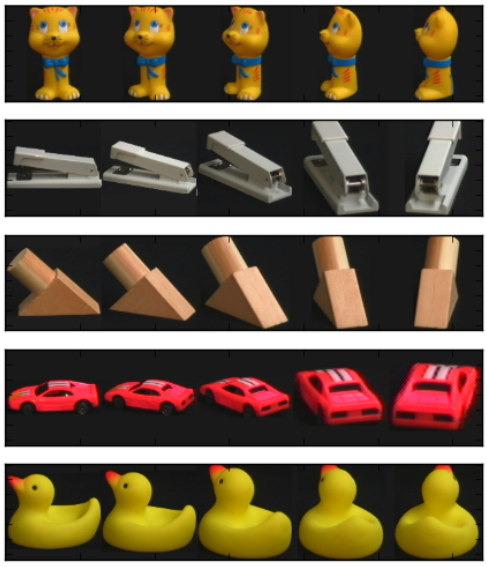

Figure 5-18. COIL-100 data set: Five example objects rotated through five angles. The COIL dataset consists of 100 objects rotated 5 degrees at time for a total of 72 images per object.

- Second, we create a model for each object separately using both a random policy and divergence-to-go. We compare both policies in the learning of the models and the final quality of each model.
- Third, we combine all the transition models except one, and learn a dtg policy for the missing model. We then compare the new dtg policy in learning an as yet unseen new object with that of a random policy by comparing the predictions of each model.

5.4.6.1 Learning a representation

A 3-dimensional representation for the coil-100 dataset was learned using a deep convolutional variational autoencoder (VAE) [73]. Autoencoders are neural networks which learn, typically through a bottleneck layer, to reconstruct the input data. The bottleneck layer is a layer with a small number of nodes—this allows only the information most essential to reconstruction to be passed onto the decoder layer. Typically the

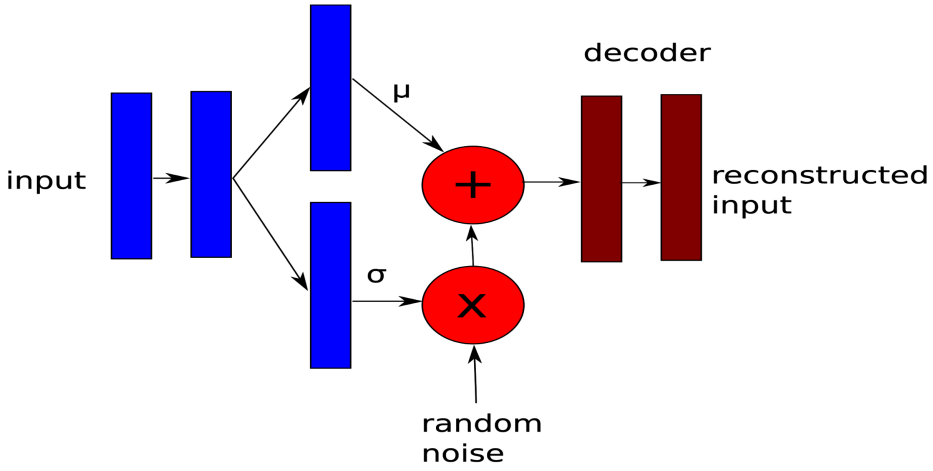

Figure 5-19. Variational (Gaussian) autoencoder architecture. The architecture consists of an encoder, a reparametrization layer, and a decoder. The variational (bottleneck) layer consists of two parallel layers which learn the mean and variance of the data's representation in the reparametrization layer. A regularization is imposed which attempts to minimize the KL divergence between some Gaussian prior and the reparametrization layer mean and variance.

reconstruction error, which is used to optimize the network weights, is measured using mean squared error (MSE), although other loss functions, such as correntropy [74], have been used. A variational autoencoder (Figure 5-19) is an autoencoder which uses a special kind of regularization on the bottleneck layer. This variational regularizer minimizes the KL divergence between the (reduced-dimensional) data and some Gaussian prior.

The VAE was trained using the entire COIL-100 dataset and a Gaussian prior with standard deviation of 0.5. As shown in Figure 5-19, the VAE consists of an encoder, followed by a variational layer, followed by a decoder. The encoder consists of four convolutional layers followed by an intermediate dense layer. The decoder consists of two intermediate dense layers followed by four deconvolutional layers. Each convolutional and deconvolutional layer consists of 64 filters. The first two convolutional layers have a stride of (2,2), while the last two have a stride of (1,1). The intermediate layers on either side of the variational layer consist of 128 nodes. The second decoder intermediate

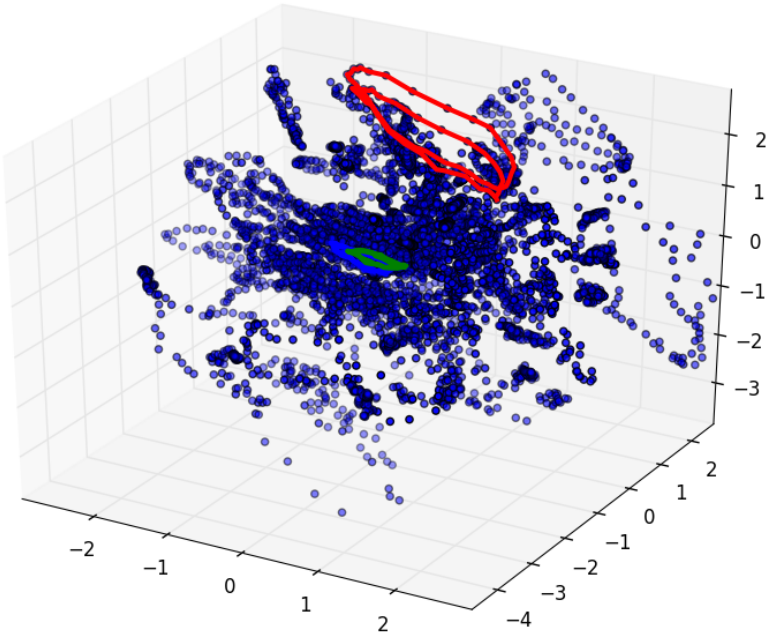

Figure 5-20. Three-dimensional representation of the COIL-100 dataset learned by a variational autoencoder. The three (blue, red, and green) rings indicate the trajectory of three rotating objects (objects 0, 2, 21) as they traverse the reduced-dimensional space.

layer consists of 16384 nodes. All layers, with the exception of the final deconvolutional layer, are followed by ReLU non-linearities. The final layer is followed by a sigmoid non-linearity. The network was trained using the rmsprop [75] algorithm.

A three-dimensional scatter plot, along with the paths of objects 0, 2, and 21 are shown in Figure 5-20. The objects, as they rotate around the turntable, tend to make loops, or double loops, within the encoded 3-dimensional space. The double-loops occur because many objects tend to look similar when rotated 180 degrees. The majority of the objects reside near the center of the 3-d representation with overlapping or nearly overlapping paths. An SVM, trained on each object at angles $0°, 90°, 180°$, and $270°$, achieves only a 66% classification rate for the rest of the data.

We created an environment using the COIL-100 dataset and its learned lower-dimensional representation. The state space consists of the three-dimensional

representations of each image in the dataset, excepting the images corresponding to the angles $0°, 90°, 180°$, and $270°$. The action set consists of the actions $A = \{-34, 33, \ldots, 0, \ldots, 33, 34\}$. Suppose the images for each object are numbered $0, 1, 2, \ldots, 67$, and suppose the current image selected is the image numbered I. Then each action a induces a transition to a state corresponding to the image $(I + a) \mod 68$, where $\mod$ is the modulo operator. Action 0 corresponds to randomly changing to a different object and angle.

5.4.6.2 Learning the models

A model for each of the 100 objects was created by building a transition model using both the divergence-to-go policy and the random policy.

The divergence-to-go policy used the following learning parameters: a learning rate $\alpha = 0.1$, a discount factor $\gamma = 0.9$, and an initial divergence multiplier $\kappa = 0.1$. Max divergence was computed by taking the maximum divergence of 100 random iterations. The following transition model parameters were selected as: state transition kernel size $\sigma = 0.1$, similarity state size $\sigma_s = 0.25$, similarity action kernel size $\sigma_a = 6.0$.

Figure 5-21 shows the dtg and random policy divergence learning curves for each of the 100 COIL objects. The blue lines are the actual divergence computed while training using dtg; the green and red lines were computed by convolving a 100-order moving average filter with the dtg and random policy divergences respectively. For almost every object, the dtg policy curves remain above the random policy curves, indicating that the dtg policy is visiting states of higher divergence, i.e., it is learning the model for the object more quickly. Figure 5-22 shows the state histograms after training using dtg for 10,000 iterations.

Table 5-4 shows the total divergence collected during training for each of the 100 objects and for both the DTG policy and random policy. Furthermore, the table shows the maximum divergence experienced for each object. The DTG policy collects more divergence than the random policy for 98 of the 100 objects. The two objects for which

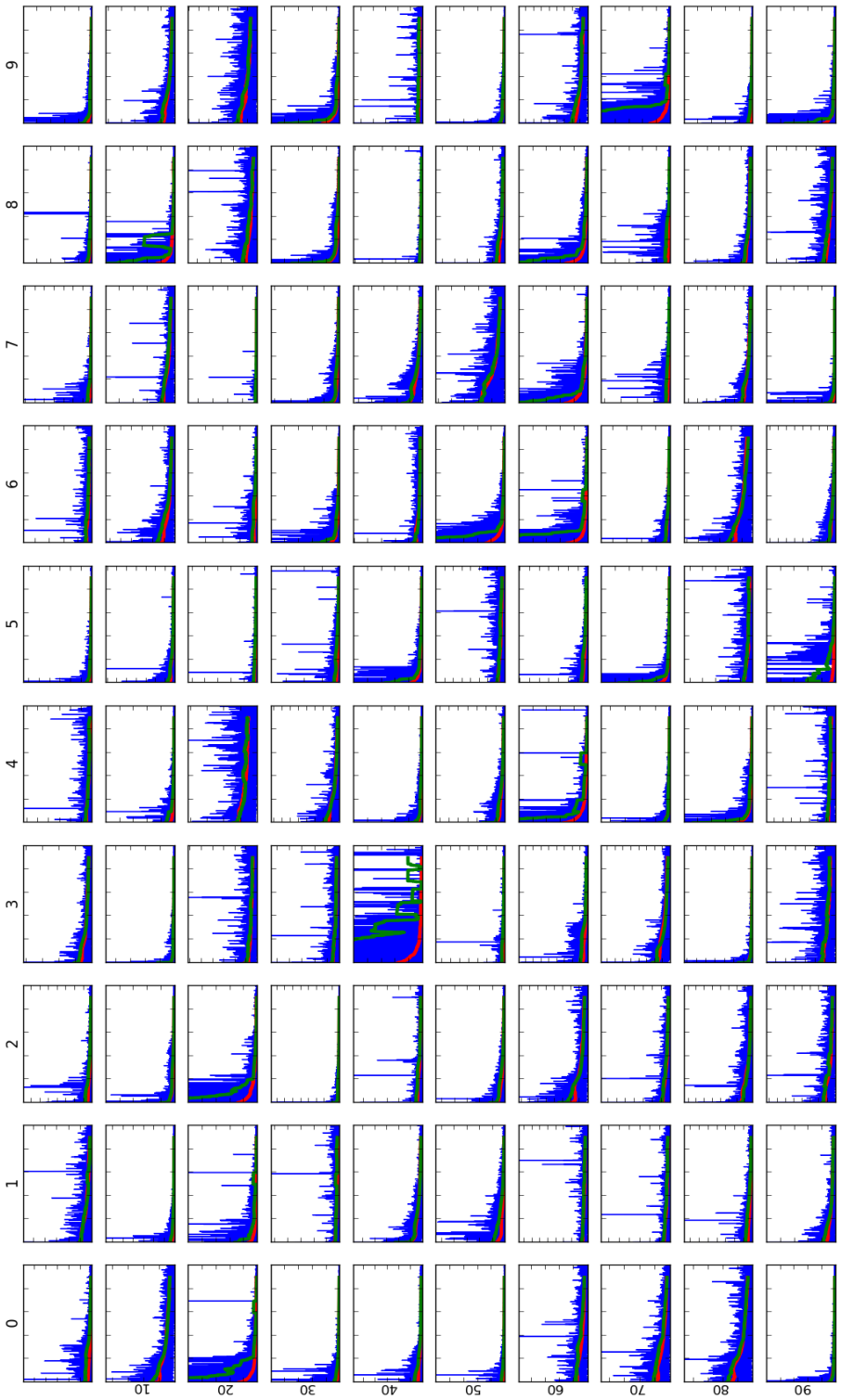

Figure 5-21. Learning models using dtg. The blue lines are the divergence obtained over 10,000 iterations while learning the COIL-100 models. The green and red lines are obtained by convolving the dtg and random (respectively) divergence curves with a 100-order moving average filter.

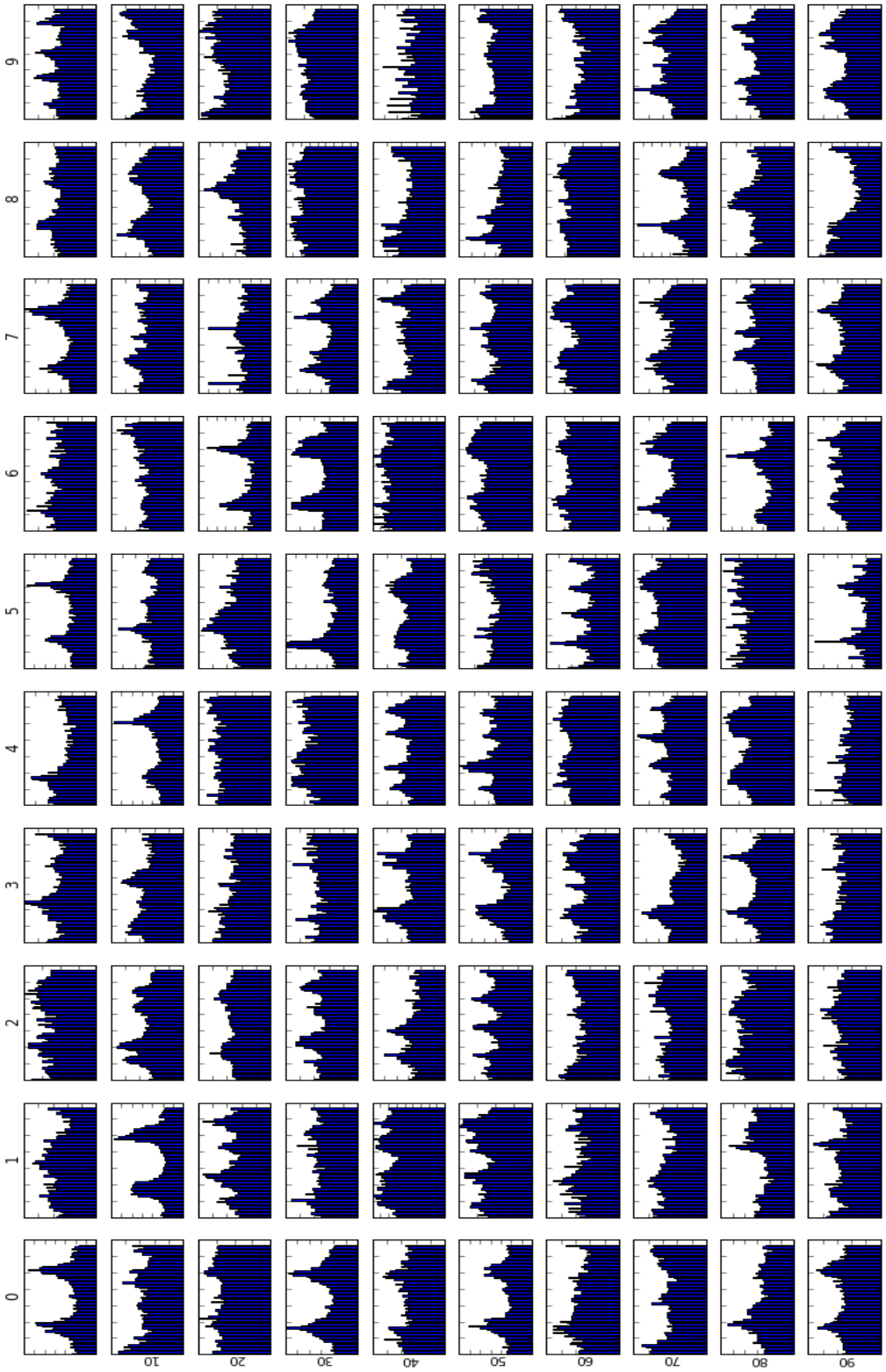

Figure 5-22. Histogram of states after exploring each object for 10,000 iterations.

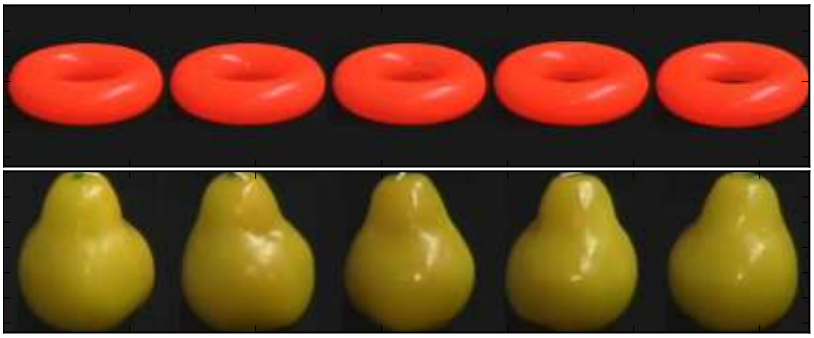

Figure 5-23. Donut and pear objects displayed at $0°$, $60°$, $120°$, $180°$, and $240°$

the random policy collected more divergence were objects 46 and 82 which correspond to the donut and pear objects (Fig 5-23). All images for the donut object are practically identical and there is little structure in the transitions within the 3-dimensional representation (Figure 5-25). The pear object is nearly rotationally symmetric and although its 3-dimensional representation does contain structure, the average distance between the (3-dimensional) points corresponding to this object's rotations is the smallest of all the objects, i.e., the pear object moves only slightly within the 3-dimensional representation. The dtg kernel size is simply too large to effectively learn this object's structure. For both objects, the DTG policy produces a state distribution that is relatively uniform as shown in Figure 5-24. These histograms strongly resemble the state histograms generated by the random policy.

Let us consider one of the objects, the "Dristan cold box" (obj 0), to demonstrate the properties of divergence-to-go. Suppose the states corresponding to the image rotations for each object are numbered sequentially from 1 to 68, where 1 corresponds to $0°$ and 68 corresponds to $355°$. Figure 5-26 compares histograms of the states visited after training for 10,000 iterations using both the divergence-to-go and random policies. The DTG policy was run using the parameters listed above. There are clearly two peaks in the states visited, centered around states 19 and 52. To understand why, consider

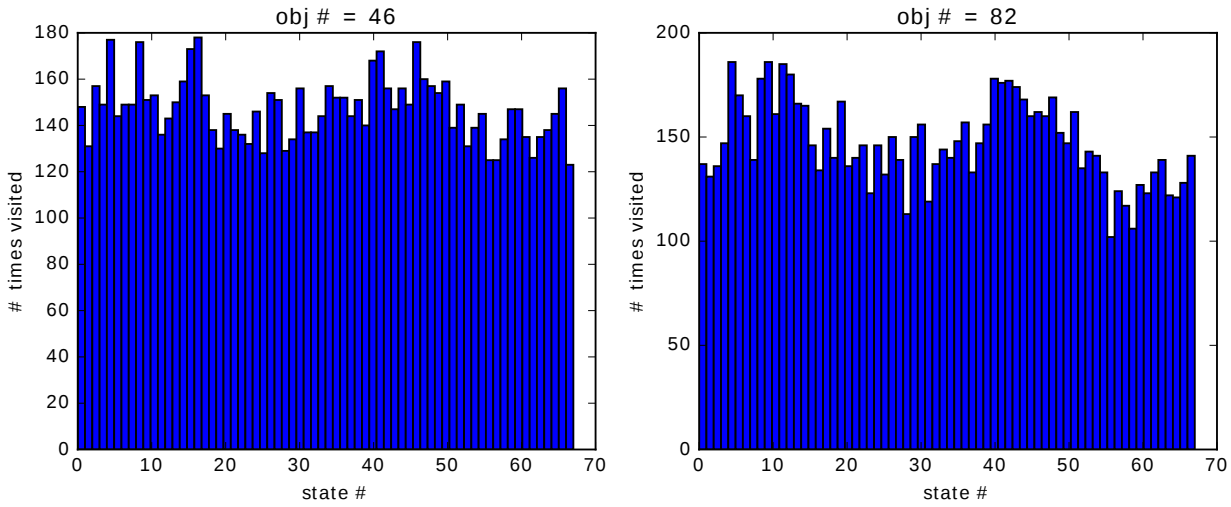

Figure 5-24. State histograms for objects 46 and 82 (donut and pear). These are the two objects for which the random policy collected more divergence than the dtg policy. For these objects, the dtg policy generates nearly uniform state histograms.

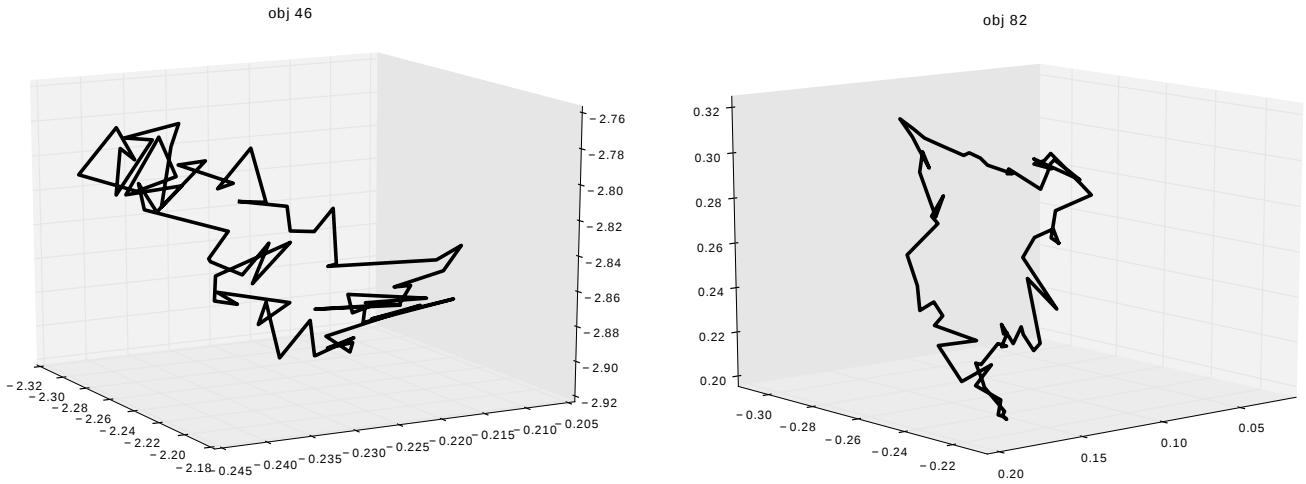

Figure 5-25. State trajectories for objects 46 and 82—the donut and the pear.

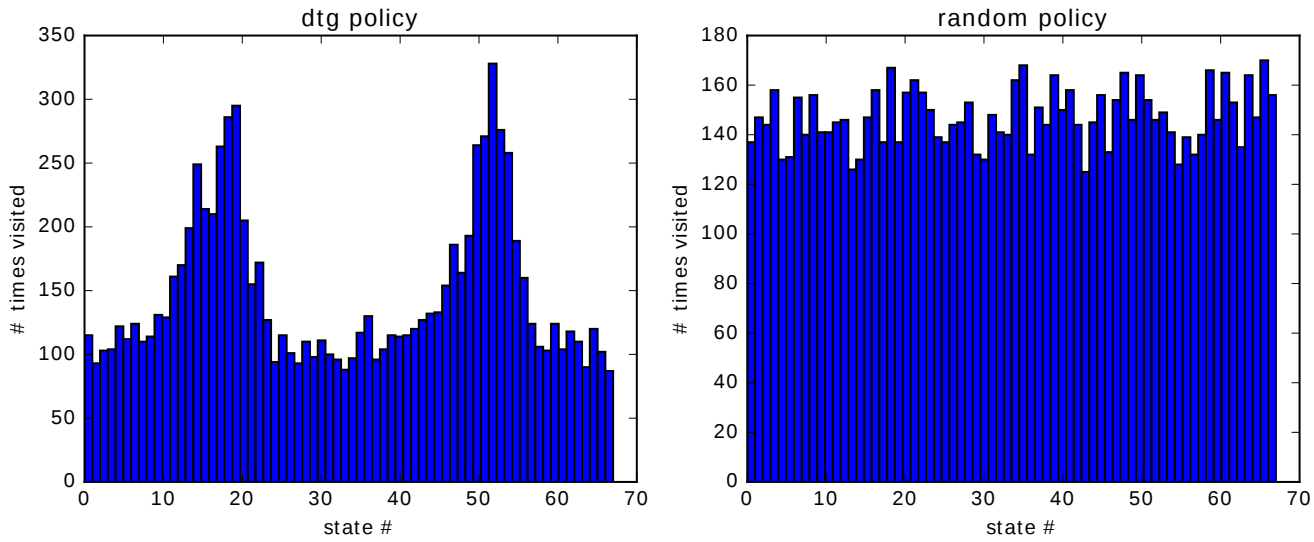

Figure 5-26. Object 0 (Dristan cold) object state histogram. Number of times each state was visited after training for 10,000 iterations using the dtg policy (left) and the random policy (right).

Figure 5-27. This plot shows the trajectory of the rotating Dristan cold box object images in the 3-dimensional space.

The trajectory forms a double loop which begins to separate at states 12 and 47. This is where the state distribution begins to increase. It then drastically separates around states 19 and 52 which are marked by a blue and green star respectively. These states correspond to the peak in the state distribution. The loop rejoins around states 25 and 60, which is where the state distrubition begins to decrease.

For the states where the loops diverge, in order to effectively learn the transition model, the divergence-to-go policy must visit these states more frequently. That is, the transition model can easily generalize in the areas of the state space where the two loop's transitions agree; however, more visits are required to learn the difference between the transitions for which two loops diverge. Although we are only examining this one object, its properties are representative of many other objects in the COIL-100 datset. In fact, due to similar symmetry, many of the objects exhibit a similar state

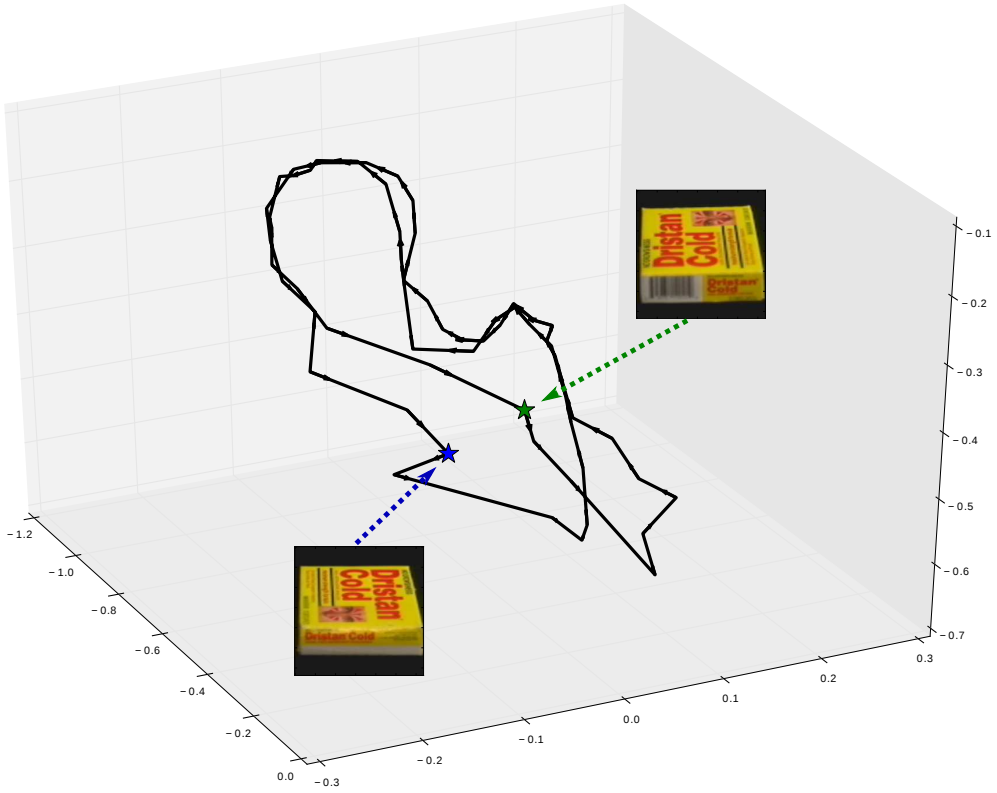

Figure 5-27. Object 0 (Dristan cold) object double loop. Trajectory of the rotating "Dristan cold" box takes as it traverses the reduced 3-dimensional space. The blue and green stars represent states 19 and 52 which correspond to the peaks of the state histograms.

histogram. Most of these objects are rectangular in shape, such as the cigarette box, or the race car.

Recall the parameter κ which we defined as a multiplier for the initial divergence-to-go according to

$$D_0 = \frac{\kappa D_{max}}{1-\gamma}.$$

The initial dtg D_0 controls the "evenness" of exploration, with high values encouraging even exploration over the space, and low values encouraging immediate exploration of areas of high divergence. For $\kappa = 1$, if the max divergence is experienced at a state, the dtg estimate for that state stays constant. For lower values of κ, if the max divergence is

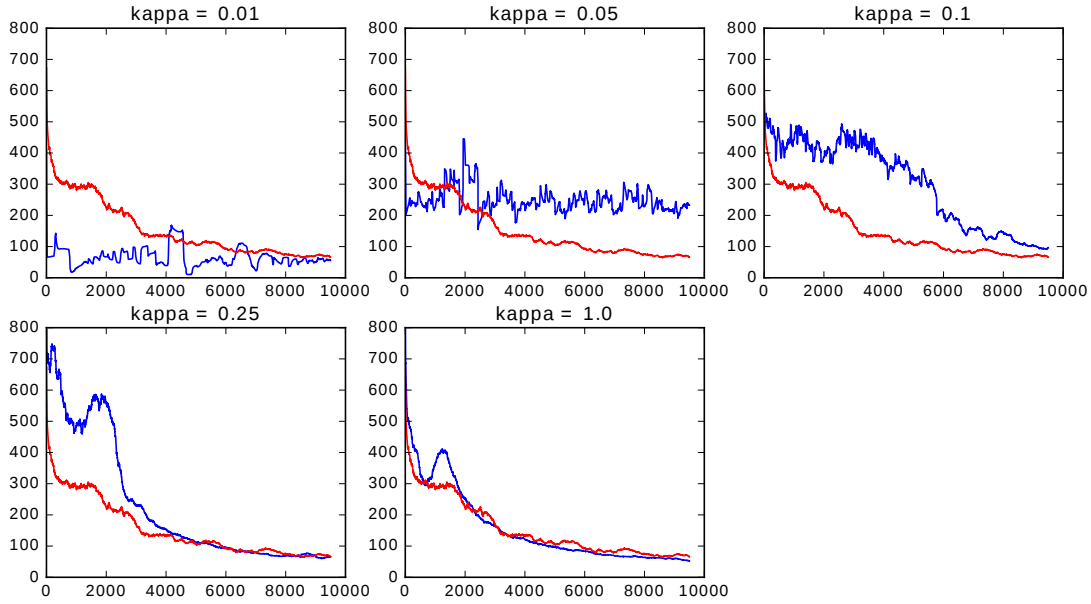

Figure 5-28. Comparison of (object 0) divergence learning curves for various values of κ. The divergence was recorded for 10000 iterations of training and the learning curve was obtained by convolving the divergence with a 100-order moving average filter.

experienced, the dtg estimate actually *increases*, causing states in the proximity of the update to be immediately more valuable than currently unvisited states.

We ran experiments comparing different values of κ while using the parameters $\alpha = 0.1$ and $\gamma = 0.9$. During training, the divergence was recorded for each transition over 10000 iterations. A learning curve was then obtained by convolving the divergence with a 100-order moving average filter. Figure 5-28 compares the divergence learning curve for various values of κ for both the dtg and random policies. The blue curve corresponds to the dtg policy while the red curve corresponds to the random policy.

For $\kappa = 0.01$, the initial dtg D_0 is too small to facilitate effective learning, with the dtg policy divergence curve staying below the random policy divergence curve over almost all iterations. This occurs because, with κ set this low, the agent attempts to thoroughly explore every area of the state space before moving onto another area. As

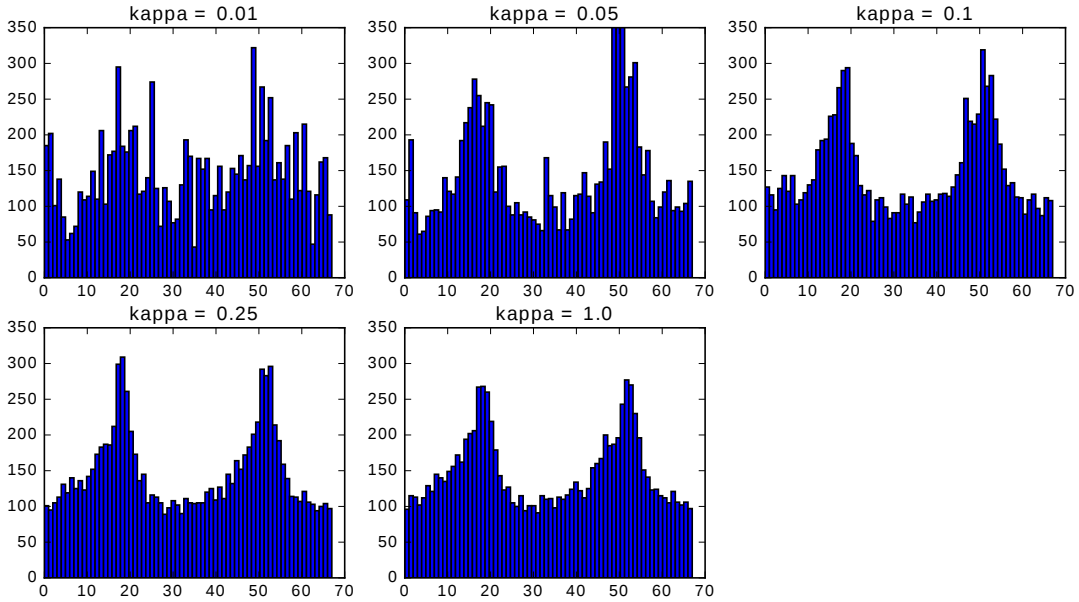

Figure 5-29. Comparison of (object 0) state histograms for various values of κ. The histograms were computed from the states visited after training for 10,000 iterations.

κ increases to 0.1 the dtg policy divergence curve consistently stays above the random policy divergence curve as the policy begins to focus on high divergence areas of the state space.

As κ increases, by $\kappa = 0.25$ a distinctive hump begins to form at around iteration 2000. This "hump" is characteristic of the dtg learning curve for almost all of the objects in this environment. The drop in divergence from iterations 0 to 1000 is due to exploring the entire state space. The subsequent rise at the "hump" is due to the agent returning to explore areas of high divergence. This "hump" is indicative of the information available in the environment that the random policy ignores.

Figure 5-29 shows the histogram of all the states visited after training using the divergence-to-go for 10,000 iterations. The same values of κ shown in Figure 5-28 were used. The histogram corresponding to $\kappa = 0.01$ appears disorganized with a few individual states receiving much more attention than the rest of the states. As

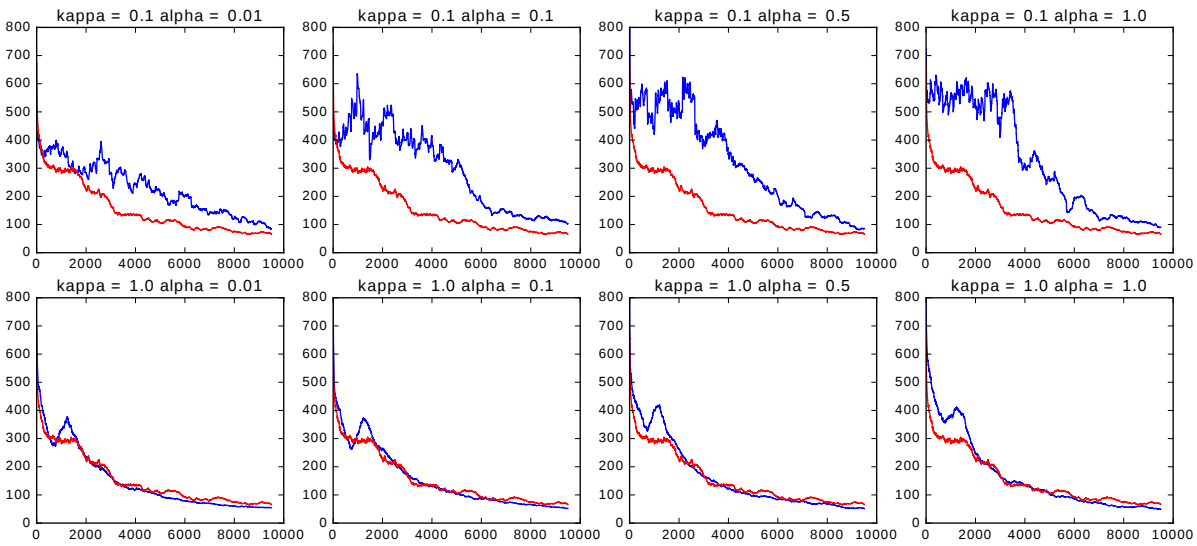

Figure 5-30. Comparison of (object 0) divergence learning curves for various values of learning rate α at κ values of 0.1 and 1.0.

κ is increased, the two peaks begin to form. The height of the peaks decrease as κ approaches 1.0. For values of $\kappa > 1$, the divergence curves and histograms do not vary much from $\kappa = 1$, even for extremely large values of κ.

We ran further experiments comparing different values of the learning rate α for different values of κ and a discount factor $\gamma = 0.9$. Once again, during training, the divergence was recorded for each transition over 10000 iterations. A learning curve was again obtained by convolving the divergence with a 100-order moving average filter. Figure 5-30 compares the divergence learning curve for various values of α and κ values of 0.1 and 1.0 for both the dtg and random policies. Once again, the blue curve corresponds to the dtg policy while the red curve corresponds to the random policy.

For the very small learning rate of $\alpha = 0.01$ and $\kappa = 0.1$, the dtg policy learning curve is only slightly above the one corresponding to the random policy. As α increases up to 1.0, the dtg learning curve gradually separates from the random policy learning curve. The same holds true for $\kappa = 1.0$, but to a lesser extent.

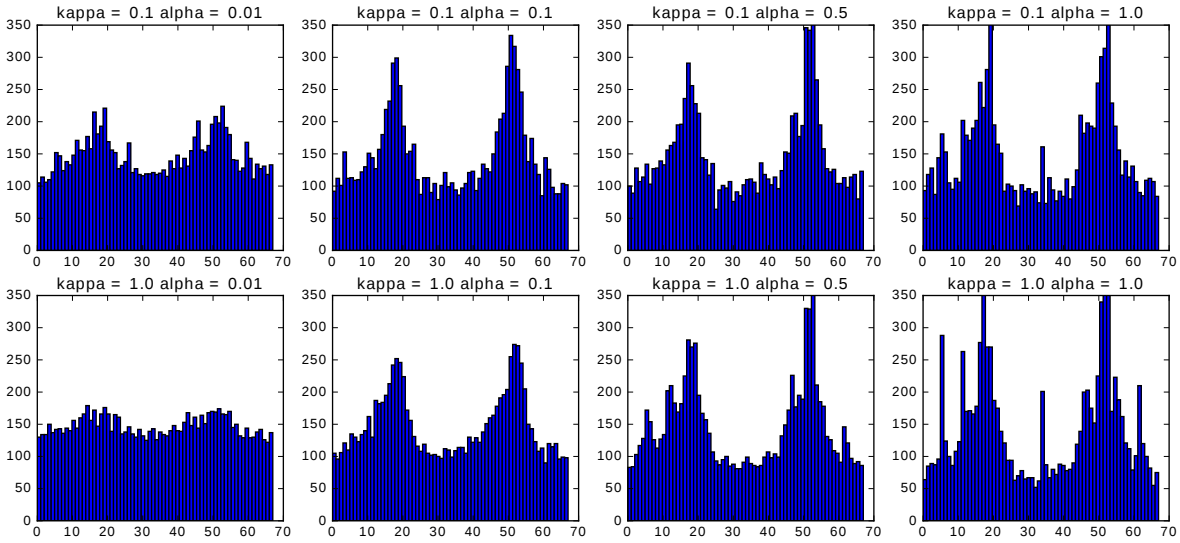

Figure 5-31. Comparison of (object 0) state histograms for various values of learning rate α at κ values of 0.1 and 1.0.

Figure 5-31 shows a histogam of all the states visited after training using divergence-to-go for 10,000 iterations. The same values of α and κ shown in Figure 5-30 were used. As α is increased, the states corresponding to high divergence clearly become more emphasized. The histogram corresponding to $\kappa = 1.0$ and $\alpha = 0.01$ is nearly uniform.

5.4.6.3 Learning a policy

After learning each of the models separately, we combined them into a single aggregated model. We then trained a dtg policy for each model by first removing that model from the aggregated model, and then using dtg to build the model from scratch. For this purpose, we again used the training parameters of learning rate $\alpha = 0.1$, discount factor $\gamma = 0.9$, and $\kappa = 0.1$. The same kernel parameters as above were used.

Since the goal is to learn a policy from any initial state for any condition of the model, we reset both the model and the environment every 25 iterations. In this way, we learn a policy which builds the model the most quickly in 25 steps. Figure 5-32 shows the divergence collected over 25 iterations as the policy is learned. It also shows the divergence that the random policy collects. The blue and red lines represent divergence

for the dtg and random policies, respectively. Furthermore, both divergences were convolved with a 100-order moving average filter which produced the green and yellow lines for the dtg and random policies, respectively.

Finally, we compared the learned dtg policies with the random policy. The dtg policy attempts to learn the transitions which are most different from the rest of the models. Thus, it a builds a model using transitions which are most discriminative for the object.

For each object, the COIL environment was initialized to a random state and then followed one of the policies. For each step, each of the 100 models was compared to predict which object was being viewed. Winner-take-all voting was used to determine what the predicted object was at each step. After 25 iterations, the environment was reset to a random initial state.

Table 5-5 shows the number of steps required to reach of 0.95 classification rate for both the dtg and random policies. By this measure, the dtg policy outperforms the random policy for 82/100 objects, ties the random policy for 14/100 objects, and only performs worse than the random policy for 4 objects. The objects for which the dtg policy performs worse are by only one or two steps. The random policy is unable to reach of classification rate of 0.95 for objects 8, 83, and 94. Ignoring the objects for which the random policy does not reach 0.95, the largest difference between the two policies is object 5, where the dtg policy correctly classifies at a rate of 0.95 in 5 steps, while the random policy requires 23 steps, for a difference of 18 steps.

Figure 5-33 shows the fraction of correctly classified objects at each iteration. The blue lines represent the dtg policy and the red lines represent the random policy.

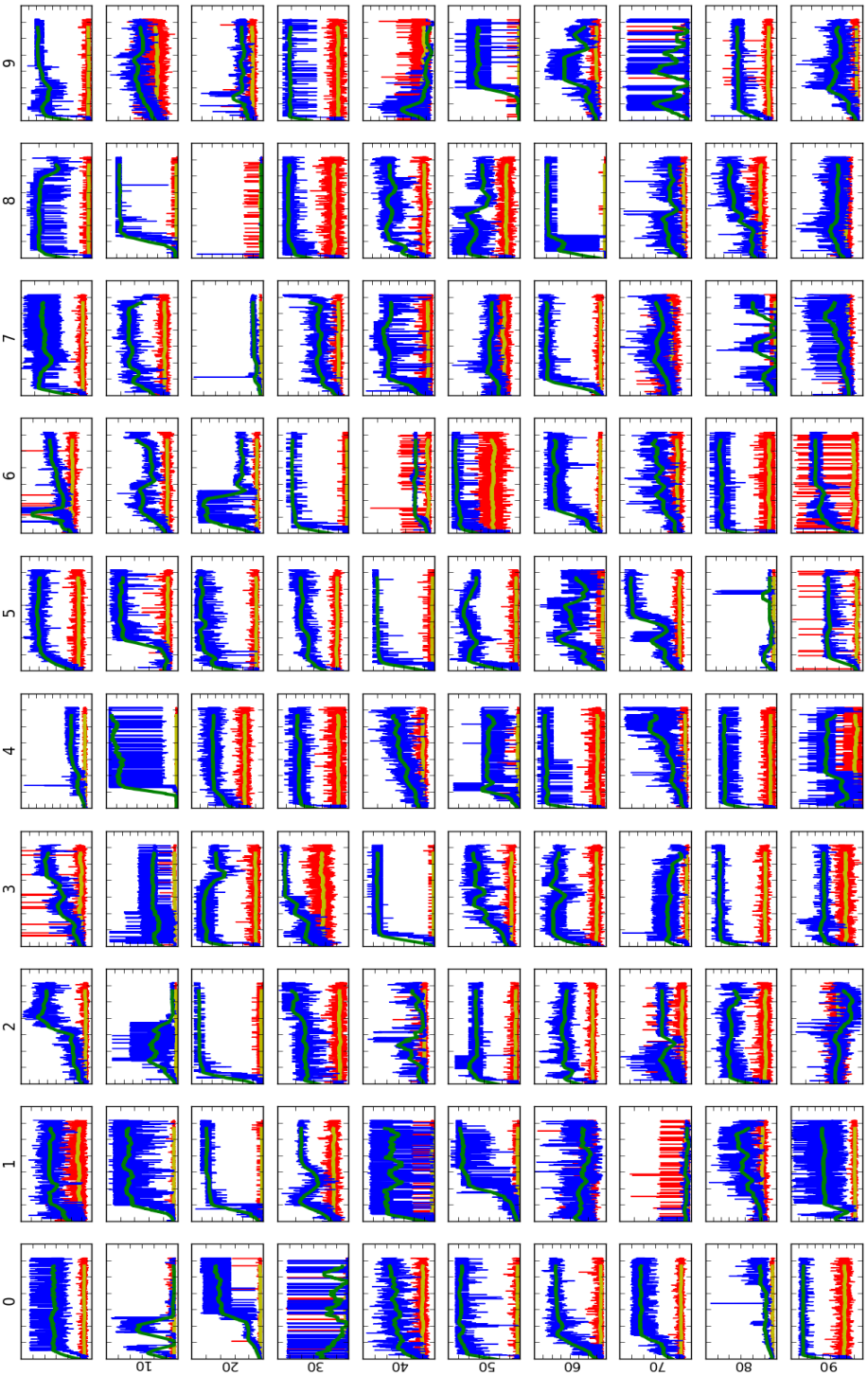

Figure 5-32. Learning a dtg policy. The divegence is recorded over 25 iterations after which the transition model is reset. The blue and red lines correspond to the divergence for the dtg and random policies. The green and yellow lines are obtained by convolving the dtg and random (respectively) divergence with a 100-order moving average filter.

Table 5-4. Collected divergence while learning a transition model over 10,000 iterations for each object in the COIL-100 data set. The DTG and random policies are compared

	Div Sum DTG policy	Div Sum Rand Policy	Max Div		Div Sum DTG Policy	Div Sum Rand Policy	Max Div
0	28667.20	17052.11	118.46	50	38153.76	28872.60	506.21
1	2292.63	2284.80	5.73	51	76285.44	50254.86	270.21
2	15912.08	11767.60	78.93	52	67557.04	48623.21	271.65
3	10039.31	9586.57	9.74	53	16371.24	11783.26	152.65
4	1702.78	1598.92	6.34	54	50667.12	34647.68	66.76
5	32269.58	27632.04	323.52	55	1402.46	1224.86	3.99
6	1403.79	1218.64	4.83	56	469569.29	116722.92	10000.00
7	39032.96	35138.52	361.55	57	4310.05	4215.77	5.15
8	59351.34	51037.57	573.63	58	16952.53	13294.61	20.68
9	73575.82	47643.66	756.20	59	54395.59	53085.71	529.37
10	15725.09	14893.37	22.01	60	12547.81	10612.51	24.26
11	27537.83	20770.24	331.51	61	4447.34	3985.16	17.27
12	87606.11	62437.05	460.56	62	9812.48	9228.68	16.58
13	78064.89	39475.15	179.75	63	40217.01	28486.30	177.02
14	65647.73	43707.26	10000.00	64	799759.76	144052.07	10000.00
15	24833.58	21076.49	143.08	65	29818.18	24663.39	141.98
16	16761.00	14052.57	31.83	66	1077020.93	119918.69	10000.00
17	12874.87	11646.13	24.75	67	455392.08	151204.17	10000.00
18	386400.09	106906.19	10000.00	68	458909.74	133222.94	10000.00
19	11621.41	11218.21	18.85	69	10547.19	9260.78	17.80
20	909907.92	118746.95	10000.00	70	14010.57	11801.45	22.56
21	151865.89	69922.58	10000.00	71	3784.74	3328.03	13.49
22	701160.45	133350.13	10000.00	72	2205.29	1975.06	7.69
23	1465.71	1390.45	3.07	73	9544.58	8166.14	9.37
24	1972.45	1874.99	2.06	74	62943.77	50631.04	459.37
25	13812.63	10335.16	105.19	75	219247.98	72765.90	10000.00
26	35232.15	21534.86	198.65	76	35326.54	34325.17	238.15
27	24381.14	14718.58	323.98	77	32987.04	20744.84	210.36
28	3254.60	2808.39	5.75	78	38340.85	17277.51	249.35
29	2853.98	2635.82	2.51	79	1550528.09	119911.99	10000.00
30	50617.15	39456.28	600.75	80	7917.11	7311.26	9.30
31	616.37	562.05	2.62	81	20460.33	15201.49	76.56
32	44185.39	29069.47	66.78	82	1420.64	1539.28	4.25
33	349.70	345.90	1.09	83	53955.07	36993.72	365.06
34	6761.06	6492.86	10.23	84	269431.18	103403.17	10000.00
35	35240.97	21823.29	191.99	85	1031.93	957.58	2.74
36	147229.36	80121.50	10000.00	86	10057.83	9994.29	12.54
37	51473.02	40581.32	210.51	87	15942.43	13687.71	26.99
38	235954.49	109916.45	10000.00	88	9347.75	8603.25	25.55
39	278407.71	113528.18	10000.00	89	51184.40	35816.59	729.68
40	23845.63	17323.11	116.46	90	48601.62	44338.12	819.85
41	46329.57	37756.39	87.29	91	17142.15	15152.91	26.72
42	6789.29	5904.29	29.17	92	3682.58	3238.70	8.27
43	2333530.52	148589.09	10000.00	93	851.28	773.97	1.38
44	95161.53	48545.10	686.81	94	3332.27	2909.35	7.86
45	181558.17	81043.93	10000.00	95	190283.39	48663.79	10000.00
46	1242.67	1244.40	5.04	96	46062.98	36060.01	56.10
47	44406.42	41774.37	108.13	97	54654.19	30638.84	641.47
48	9387.26	8346.72	102.76	98	3324.38	2826.89	6.87
49	2953.97	2313.09	11.12	99	213089.24	75277.15	10000.00

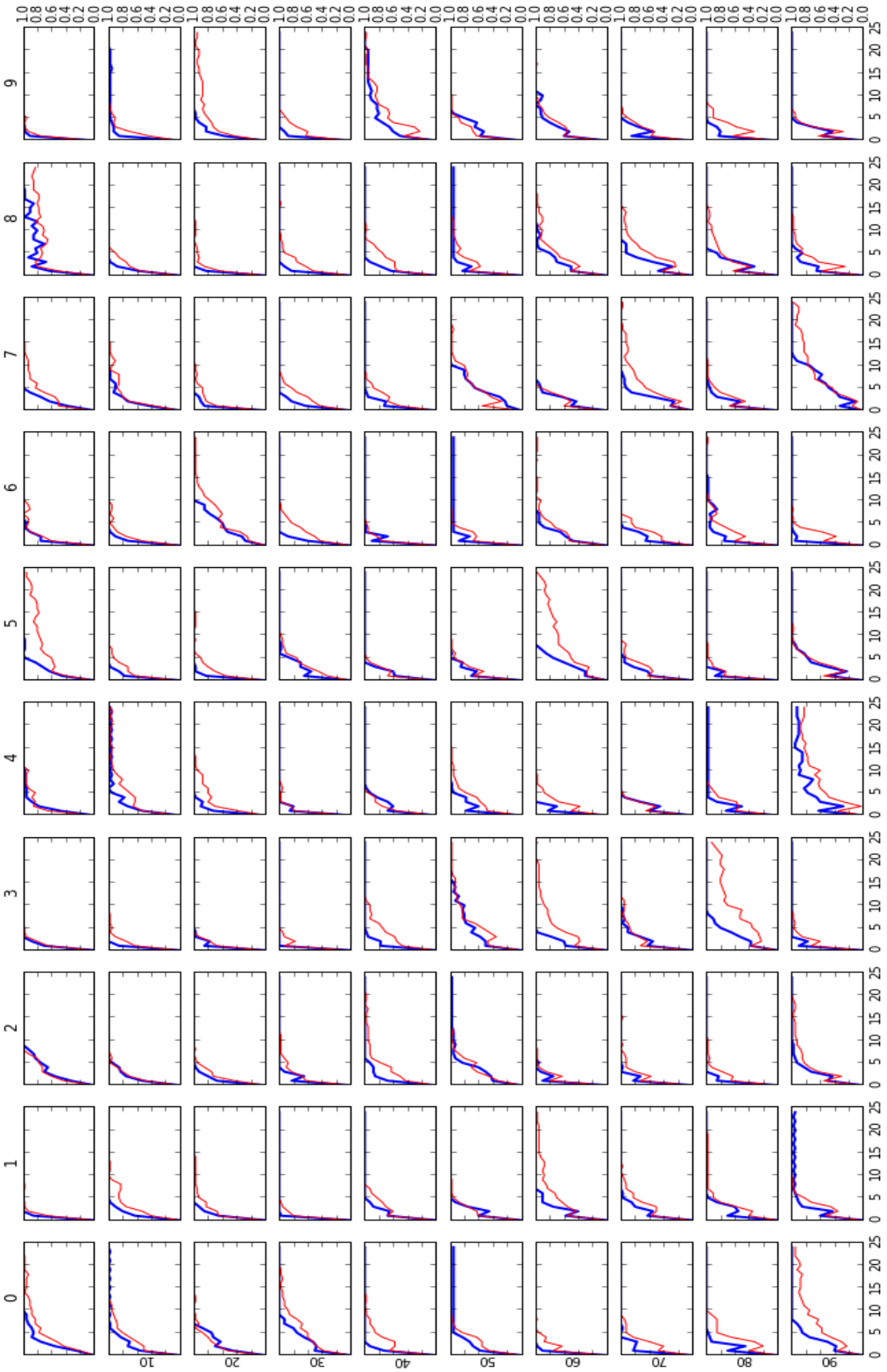

Figure 5-33. COIL classification results

Table 5-5. COIL-100 environment object identification. Comparison between dtg and random policies for the number of steps required to reach 0.95 classification rate for each object.

Obj #	DTG	Random	Obj #	DTG	Random	Obj #	DTG	Random	Obj #	DTG	Random
0	8	19	25	2	6	50	5	7	75	4	9
1	2	3	26	10	14	51	5	5	76	4	7
2	9	8	27	4	6	52	10	8	77	6	18
3	3	3	28	2	5	53	15	17	78	8	15
4	4	11	29	5	21	54	5	9	79	5	6
5	5	23	30	9	13	55	5	7	80	4	10
6	4	10	31	1	3	56	3	5	81	5	7
7	5	12	32	4	7	57	10	12	82	3	5
8	17	$>$25	33	1	4	58	4	9	83	8	$>$25
9	2	3	34	3	3	59	6	6	84	4	7
10	6	10	35	6	8	60	2	5	85	3	5
11	4	9	36	3	8	61	7	15	86	10	10
12	5	5	37	4	7	62	4	4	87	5	7
13	2	4	38	3	6	63	4	14	88	6	12
14	8	11	39	3	6	64	3	7	89	4	8
15	3	6	40	3	9	65	8	24	90	8	22
16	3	7	41	5	8	66	5	11	91	6	6
17	7	12	42	6	10	67	6	6	92	6	11
18	3	6	43	3	11	68	10	14	93	3	5
19	4	6	44	6	5	69	11	10	94	25	$>$25
20	7	8	45	4	5	70	4	8	95	8	8
21	4	6	46	3	3	71	5	9	96	3	5
22	4	5	47	3	7	72	3	7	97	12	24
23	3	4	48	4	8	73	6	7	98	6	8
24	4	10	49	13	14	74	4	4	99	5	5

CHAPTER 6
CONCLUSION

Reinforcement learning has been shown to be a extremely useful strategy for learning to exploit previously unknown environments within the framework of the perception-action-reward cycle. Despite this, it has many shortcomings. These problems include the “curse of dimensionality”, the difficulty in learning value functions and policies for continuous state and action spaces, and the problem of effective exploration. The goal of this dissertation was to introduce methods which could alleviate these problems.

In this dissertation, we have developed and tested a comprehensive framework for learning representations of environments and effectively exploring them using information-theoretic concepts. We believe this framework is an effective starting point for any pracitioner attempting to design an agent which can effectively explore and exploit complex environments.

6.1 Summary

In Chapter 2 we introduced the background necessary to implement the work shown in this dissertation. We introduced reinforcement learning within the framework of Markov decision processes, information-theoretic learning, and function approximation using kernel least mean squares.

In Chapter 3, the method of kernel temporal differences was presented. We introduced the action kernel and compared the effect of various kernel sizes on learning within the mountain car environment. Furthermore, we introduced the method of experience replay in the RKHS within the framework of q-ktd. We showed the effect of experience replay on learning within the mountain car environment, and how learning changes while varying batch size and the frequency of update. More specifically, we showed that increasing the frequency of replay updates significantly improves the speed of convergence, while increasing the replay batch size increases the stability.

In Chapter 4, we presented a method to learn new representations for an environment's state space using metric learning and nearest neighbor approximation within the framework of tabular Q-learning. The approach was to learn a weighting for state features such that neighboring states represent similar situations. This was achieved by applying centered alignment metric learning (CAML) so that states with similar Q-values are closer while states with different Q-values are far away. We showed, using the atari game of "Frogger," that this approach is extremely effective in speeding up learning. In fact, this approach made learning possible in situations where conventional tabular Q-learning was incapable of learning such a large state space.

Chapter 5 introduced divergence-to-go, an approach to directed exploration within the reinforcement learning and perception-action-reward cycle framework. The divergence-to-go framework iteratively builds a model of the environment and then selects actions within the environment so as to most quickly improve that model. It accomplishes this by finding the actions which maximize the divergence of the model, i.e., the actions which produce the most divergence between old and new models.

To discover good policies, RL methods need the agent to explore the entire state space, which may prove impossible—especially for continuous spaces. Divergence-to-go allows a cursory exploration of a state space, only directing more thorough examination to places of interest. Divergence-to-go guides exploration to places in the enviroment where dynamics are different from their surroundings. Furthermore, many environments, and indeed the real world, are not stationary, i.e., the dynamics of these environments change over time. If the dynamics of an area of the environment has changed, divergence-to-go directs the agent to explore these areas more thoroughly. We showed divergence-to-go was effective in directing exploration and learning models for various maze problems, the mountain car problem, and the COIL-100 environment. For the COIL environment, divergence-to-go was shown to be successful in learning a policy which finds the most discriminative state-actions.

6.2 Future Work

While we have presented a complete framework for exploration, there remains work to be done on determining the tradeoff between exploration directed by divergence-to-go and reward exploitation directed by Q-learning. Actions are selected according to

$$\underset{a}{\operatorname{argmax}}\, Q(x, a) + \lambda dtg(x, a), \tag{6–1}$$

but there is currently no principled way to select the tradeoff parameter λ. This is a difficult problem, because the divergence-to-go is an information-theoretic measure, while the Q-value is a measure of rewards. A promising line of research based on the value of information (VOI) is being investigated [76]. This goal of this research to quantify the value, in terms of costs or rewards, in gaining a piece of information. Since Equation (6–1) contains Q, an expected sum of rewards, and dtg, an expected sum of information quantities, this approach seems encouraging.

Divergence-to-go as defined in Equation (5–32) can be formulated asa differentiable cost function. Differentiability is a requirement that makes it easier to use deep neural networks for optimization. Thus, a promising line of research for future work would be to apply the exploration schemes proposed here to deep reinforcement learning. Divergence-to-go deep reinforcement learning would combine the power of powerful parametric value/policy functions with the environment discovery driven exploration of divergence-to-go. Also, divergence-to-go would replace other ad-hoc differentiable constraints used to force exploration—for example maximizing the entropy of the action distribution given an observation. From a philosophical perspective, divergence-to-go can also be compared to the curiosity drives in intelligent systems. Thus, we could also compare how divergence-to-go compares to other curiosity-based [24] exploration schemes in deep learning.

REFERENCES

[1] C. J. C. H. Watkins, “Learning from delayed rewards,” Ph.D. dissertation, King’s College, Cambridge, UK, May 1989.

[2] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[3] R. S. Sutton, “Learning to predict by the methods of temporal differences,” *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[4] S. Mahadevan and J. Connell, “Automatic programming of behavior-based robots using reinforcement learning,” *Artificial intelligence*, vol. 55, no. 2, pp. 311–365, 1992.

[5] R. S. Sutton and A. G. Barto, *Introduction to reinforcement learning*. MIT Press, 1998.

[6] D. P. Bertsekas, *Dynamic programming and optimal control*. Athena Scientific Belmont, MA, 1995, vol. 1, no. 2.

[7] L. Tesfatsion and K. L. Judd, *Handbook of computational economics: agent-based computational economics*. Elsevier, 2006, vol. 2.

[8] J. DiGiovanna, B. Mahmoudi, J. Fortes, J. C. Principe, and J. C. Sanchez, “Coadaptive brain–machine interface via reinforcement learning,” *Biomedical Engineering, IEEE Transactions on*, vol. 56, no. 1, pp. 54–64, 2009.

[9] B. Mahmoudi and J. C. Sanchez, “A symbiotic brain-machine interface through value-based decision making,” *PloS one*, vol. 6, no. 3, p. e14760, 2011.

[10] G. Tesauro, “Td-gammon, a self-teaching backgammon program, achieves master-level play,” *Neural computation*, vol. 6, no. 2, pp. 215–219, 1994.

[11] I. Ghory, “Reinforcement learning in board games,” *Department of Computer Science, University of Bristol, Tech. Rep*, 2004.

[12] M. A. Wiering *et al.*, “Self-play and using an expert to learn to play backgammon with temporal difference learning,” *Journal of Intelligent Learning Systems and Applications*, vol. 2, no. 02, p. 57, 2010.

[13] J. B. Pollack and A. D. Blair, “Co-evolution in the successful learning of backgammon strategy,” *Machine Learning*, vol. 32, no. 3, pp. 225–240, 1998.

[14] G. Tesauro, “Comments on co-evolution in the successful learning of backgammon strategy,” *Machine Learning*, vol. 32, no. 3, pp. 241–243, 1998.

[15] X. Yan, P. Diaconis, P. Rusmevichientong, and B. V. Roy, “Solitaire: Man versus machine,” in *Advances in Neural Information Processing Systems*, 2004, pp. 1553–1560.

[16] J. Baxter, A. Tridgell, and L. Weaver, "Knightcap: a chess program that learns by combining td (lambda) with game-tree search," *arXiv preprint cs/9901002*, 1999.

[17] J. Schaeffer, M. Hlynka, and V. Jussila, "Temporal difference learning applied to a high-performance game-playing program," in *Proceedings of the 17th international joint conference on Artificial intelligence-Volume 1*. Morgan Kaufmann Publishers Inc., 2001, pp. 529–534.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[19] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *arXiv preprint arXiv:1207.4708*, 2012.

[20] P. Stone and R. S. Sutton, "Scaling reinforcement learning toward robocup soccer," in *ICML*, vol. 1, 2001, pp. 537–544.

[21] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang, "Autonomous inverted helicopter flight via reinforcement learning," in *Experimental Robotics IX*. Springer, 2006, pp. 363–372.

[22] J. Moody and M. Saffell, "Learning to trade via direct reinforcement," *Neural Networks, IEEE Transactions on*, vol. 12, no. 4, pp. 875–889, 2001.

[23] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *arXiv preprint arXiv:1602.01783*, 2016.

[24] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel, "Variational information maximizing exploration," *arXiv preprint arXiv:1605.09674*, 2016.

[25] R. Bellman, *Dynamic Programming*, 1st ed. Princeton, NJ, USA: Princeton University Press, 1957.

[26] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. Cambridge Univ Press, 1998, vol. 1, no. 1.

[27] D. P. Bertsekas and J. N. Tsitsiklis, "Neuro-dynamic programming (optimization and neural computation series, 3)," *Athena Scientific*, vol. 7, pp. 15–23, 1996.

[28] H. Robbins and S. Monro, "A stochastic approximation method," *The Annals of Mathematical Statistics*, pp. 400–407, 1951.

[29] S. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvári, "Convergence results for single-step on-policy reinforcement-learning algorithms," *Machine Learning*, vol. 38, no. 3, pp. 287–308, 2000.

[30] T. Hester and P. Stone, "Learning and using models," in *Reinforcement Learning - State of the Art*, M. Wiering and M. van Otterlo, Eds. Springer, 2012, vol. 12, ch. 4.

[31] C. G. Atkeson, A. W. Moore, and S. Schaal, "Locally weighted learning for control," in *Lazy learning*. Springer, 1997, pp. 75–113.

[32] T. Hester and P. Stone, "Generalized model learning for reinforcement learning in factored domains," in *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*. International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 717–724.

[33] C. Boutilier, T. Dean, and S. Hanks, "Decision-theoretic planning: Structural assumptions and computational leverage," *arXiv preprint arXiv:1105.5460*, 2011.

[34] M. P. Deisenroth, *Efficient reinforcement learning using gaussian processes*. KIT Scientific Publishing, 2010, vol. 9.

[35] N. K. Jong and P. Stone, "Model-based function approximation in reinforcement learning," in *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*. ACM, 2007, p. 95.

[36] J. C. Principe, D. Xu, and J. Fisher, "Information theoretic learning," *Unsupervised adaptive filtering*, vol. 1, pp. 265–319, 2000.

[37] M. Kearns and S. Singh, "Near-optimal reinforcement learning in polynomial time," *Machine Learning*, vol. 49, no. 2-3, pp. 209–232, 2002.

[38] R. I. Brafman and M. Tennenholtz, "R-max-a general polynomial time algorithm for near-optimal reinforcement learning," *The Journal of Machine Learning Research*, vol. 3, pp. 213–231, 2003.

[39] K. Torkkola, "Feature extraction by non-parametric mutual information maximization," *Journal of machine learning research*, vol. 3, no. Mar, pp. 1415–1438, 2003.

[40] W. Liu, J. C. Principe, and S. Haykin, *Kernel Adaptive Filtering: A Comprehensive Introduction*. John Wiley & Sons, 2011, vol. 57.

[41] B. Widrow and S. D. Stearns, "Adaptive signal processing," *Englewood Cliffs, NJ, Prentice-Hall, Inc., 1985, 491 p.*, vol. 1, 1985.

[42] W. Liu, P. P. Pokharel, and J. C. Principe, "The kernel least-mean-square algorithm," *Signal Processing, IEEE Transactions on*, vol. 56, no. 2, pp. 543–554, 2008.

[43] B. Chen, S. Zhao, P. Zhu, and J. C. Príncipe, "Quantized kernel least mean square algorithm," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 1, pp. 22–32, 2012.

[44] J. Bae, L. S. Giraldo, P. Chhatbar, J. Francis, J. Sanchez, and J. Principe, "Stochastic kernel temporal difference for reinforcement learning," in *Machine Learning for Signal Processing (MLSP), 2011 IEEE International Workshop on*. IEEE, 2011, pp. 1–6.

[45] L.-J. Lin, "Reinforcement learning for robots using neural networks," DTIC Document, Tech. Rep., 1993.

[46] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[47] S. Rao, A. de Medeiros Martins, and J. C. Príncipe, "Mean shift: An information theoretic perspective," *Pattern Recognition Letters*, vol. 30, no. 3, pp. 222–230, 2009.

[48] K. Fukunaga and L. Hostetler, "The estimation of the gradient of a density function, with applications in pattern recognition," *IEEE Transactions on information theory*, vol. 21, no. 1, pp. 32–40, 1975.

[49] R. Bellman, "A markovian decision process," DTIC Document, Tech. Rep., 1957.

[50] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *The Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.

[51] G. J. Gordon, "Stable function approximation in dynamic programming," DTIC Document, Tech. Rep., 1995.

[52] C. Cortes, M. Mohri, and A. Rostamizadeh, "Algorithms for learning kernels based on centered alignment," *The Journal of Machine Learning Research*, vol. 13, pp. 795–828, 2012.

[53] A. J. Brockmeier, J. S. Choi, E. G. Kriminger, J. T. Francis, and J. C. Principe, "Neural decoding with kernel-based metric learning," *Neural Computation*, 2014.

[54] K. Fukumizu, F. R. Bach, and M. I. Jordan, "Dimensionality reduction for supervised learning with reproducing kernel hilbert spaces," *The Journal of Machine Learning Research*, vol. 5, pp. 73–99, 2004.

[55] A. Gretton, O. Bousquet, A. Smola, and B. Schölkopf, "Measuring statistical dependence with hilbert-schmidt norms," in *Algorithmic learning theory*. Springer, 2005, pp. 63–77.

[56] S. Thrun, "Learning to play the game of chess," in *Advances in Neural Information Processing Systems (NIPS) 7*, G. Tesauro, D. Touretzky, and T. Leen, Eds. Cambridge, MA: MIT Press, 1995.

[57] J. Baxter, A. Tridgell, and L. Weaver, "Learning to play chess using temporal differences," *Machine learning*, vol. 40, no. 3, pp. 243–263, 2000.

[58] D. Ormoneit and Ś. Sen, "Kernel-based reinforcement learning," *Machine learning*, vol. 49, no. 2-3, pp. 161–178, 2002.

[59] T. Gabel and M. Riedmiller, "Cbr for state value function approximation in reinforcement learning," in *Case-Based Reasoning Research and Development*. Springer, 2005, pp. 206–221.

[60] X. Huang and J. Weng, "Covert perceptual capability development," 2005.

[61] L. C. Cobo, P. Zang, C. L. Isbell Jr, and A. L. Thomaz, "Automatic state abstraction from demonstration," in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, vol. 22, no. 1, 2011, p. 1243.

[62] S. Wintermute, "Using imagery to simplify perceptual abstraction in reinforcement learning agents," *Ann Arbor*, vol. 1001, pp. 48 109–2121, 2010.

[63] C. Szepesvári, "Algorithms for reinforcement learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 4, no. 1, pp. 1–103, 2010.

[64] J. Gittins, K. Glazebrook, and R. Weber, *Multi-armed bandit allocation indices*. Wiley Online Library, 1989.

[65] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[66] J.-Y. Audibert, R. Munos, and C. Szepesvári, "Exploration–exploitation tradeoff using variance estimates in multi-armed bandits," *Theoretical Computer Science*, vol. 410, no. 19, pp. 1876–1902, 2009.

[67] R. Ortner, "Online regret bounds for markov decision processes with deterministic transitions," in *Algorithmic Learning Theory*. Springer, 2008, pp. 123–137.

[68] N. Tishby and D. Polani, "Information theory of decisions and actions," in *Perception-Action Cycle*. Springer, 2011, pp. 601–636.

[69] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.

[70] S. A. Nene, S. K. Nayar, H. Murase *et al.*, "Columbia object image library (coil-20)," Technical report CUCS-005-96, Tech. Rep., 1996.

[71] R. Chalasani and J. C. Principe, "Deep predictive coding networks," *arXiv preprint arXiv:1301.3541*, 2013.

[72] ——, "Context dependent encoding using convolutional dynamic networks," *IEEE transactions on neural networks and learning systems*, vol. 26, no. 9, pp. 1992–2004, 2015.

[73] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[74] W. Liu, P. P. Pokharel, and J. C. Príncipe, “Correntropy: properties and applications in non-gaussian signal processing,” *IEEE Transactions on Signal Processing*, vol. 55, no. 11, pp. 5286–5298, 2007.

[75] T. Tieleman and G. Hinton, “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude,” *COURSERA: Neural Networks for Machine Learning*, vol. 4, no. 2, 2012.

[76] I. J. Sledge and J. C. Principe, “Balancing exploration and exploitation in reinforcement learning using a value of information criterion,” in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. IEEE, 2017.

BIOGRAPHICAL SKETCH

Matthew Emigh was born in Sarasota, Florida. After graduating from Sarasota High School, he joined the United States Navy. He attended the Naval Nuclear Power Training Program, and graduated with the title of Reactor Operator. After graduation, he served on the USS Abraham Lincoln for four years, and was honorably discharged after achieving the rank Petty Officer, first class.

In 2007, he enrolled at the University of Florida to pursue a degree in electrical engineering. He obtained a Bachelor of Science and Master of Science degree in electrical engineering in 2010 and 2012, respectively. Matthew received his Ph.D from the Department of Electrical and Computer Engineering in May 2017.