Exploration in Feature Space for Reinforcement Learning

Suraj Narayanan Sasikumar

Under the Supervision of
Professor Marcus Hutter

A thesis submitted for the degree of
Master of Computing (Advanced)
The Australian National University

May 2017

Declaration

This thesis is an account of the research undertaken at the Research School of Computer Science, The Australian National University, Canberra, Australia.

The work presented in this thesis represents work conducted between July 2016 and May 2017. This work has been accepted for publication as "Count-Based Exploration in Feature Space for Reinforcement Learning" at the *26th International Joint Conference on Artificial Intelligence* (IJCAI) Melbourne, Australia, August 19-25 2017 (Martin et al., 2017).

The implementation of the algorithm, and the design of the infrastructure required to empirically evaluate the algorithm, are entirely my own original work. The design of the algorithm itself was done in collaboration with my colleague Jarryd Martin.

Except when acknowledged in the customary manner, the material presented in this thesis is, to the best of my knowledge, original and has not been submitted in whole or part for a degree in any university.

Suraj Narayanan Sasikumar
25 May 2017

Supervisors:

- Professor Marcus Hutter (Australian National University)
- Tom Everitt (Australian National University)

Convenor:

- Professor John Slaney (Australian National University)

To my Friends and Family who literally made this possible.

Acknowledgments

At the culmination of two years of hard work, I would like to take this opportunity to acknowledge the role they have played in my life.

- My supervisor Marcus Hutter, whose lectures inspired me to pursue Reinforcement Learning. Hearing him talk about Mathematics and AI with childlike enthusiasm has always been an inspiration to me.
- Tom Everitt, his steadfast and professional approach to supervising my thesis helped me a great deal in making this a success.
- Jarryd Martin, my collaborator in this Project. Words cannot express the happiness that I feel, to have a friend in you. The days we toiled over broken theories and random agents were not for nothing. We did it... and it could never have been achieved without you. Your determination, motivation and quite frankly the sheer energy is something that I aspire to.
- John Aslanides, first a huge thank-you for talking your time to provide your comments and feedback on this thesis. Your rationality and cut-to-the-chase attitude. Your clarity of thought, work-ethic and passion for growth has been inspirational for me.
- Lulu Huang, for talking care of me throughout this whole endeavour, and being a wonderful roommate.
- Boris Repasky, the man with infinite rigour. All the arguments and debates we've had throughout the year has only been for the better. Sina Eghbal, for being there through tough times as an unassuming friend. Darren Lawton, for forcing me to play Basketball. My AI Labmates, Sultan Javed, Owen Cameron, Arie Slobbe, Elliot Catt, for the motivation and support.
- My father Sasikumar, sister Sumitha, for believing in me and being a vocal supporter of my decisions. My in-laws Rasmi, M Vasudevan, and Geetha, for supporting my decision to study. I miss you all very much.
- My wife Ramya Vasudevan, her sacrifices and compromises are the reason I am able to do what I want to do, and I am forever indebted. None of this would even make sense without you in my life.
- My mother Sudhamony whose guidance and sacrifices made the man I am. Through out my life she has been a constant source of moral guidance. I am a better person because of her.

Abstract

The infamous exploration-exploitation dilemma is one of the oldest and most important problems in reinforcement learning (RL). Deliberate and effective exploration is necessary for RL agents to succeed in most environments. However, until very recently even very sophisticated RL algorithms employed simple, undirected exploration strategies in large-scale RL tasks.

We introduce a new optimistic count-based exploration algorithm for RL that is feasible in high-dimensional MDPs. The success of RL algorithms in these domains depends crucially on generalization from limited training experience. Function approximation techniques enable RL agents to generalize in order to estimate the value of unvisited states, but at present few methods have achieved generalization about the agent's uncertainty regarding unvisited states. We present a new method for computing a generalized state visit-count, which allows the agent to estimate the uncertainty associated with any state.

In contrast to existing exploration techniques, our $\boldsymbol{\phi}$*-pseudocount* achieves generalization by exploiting the feature representation of the state space that is used for value function approximation. States that have less frequently observed features are deemed more uncertain. The resulting $\boldsymbol{\phi}$*-Exploration-Bonus* algorithm rewards the agent for exploring in feature space rather than in the original state space. This method is simpler and less computationally expensive than some previous proposals, and achieves near state-of-the-art results on high-dimensional RL benchmarks. In particular, we report world-class results on several notoriously difficult Atari 2600 video games, including Montezuma's Revenge.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

'No great discovery was ever made without a bold guess.'

Isaac Newton

1.1 Reinforcement Learning

Machine Learning is a field in computer science that allows computers to dynamically generate novel algorithms that otherwise cannot be explicitly programmed. These algorithms, called *hypotheses*, generalize patterns and regularities from observed real-world data using statistical techniques (Bishop, 2007). *Reinforcement Learning* (RL) is a field of machine learning that deals with optimal sequential decision making in an unknown environment with no explicitly labelled training data. The RL framework is one of the fundamental models that best describes how intelligent beings interact with their world to achieve a *goal*. An RL algorithm is given agency to interact with its surroundings, and is aptly called an *agent*. The world with which the agent interacts is called its *environment*. The *unsupervised* nature of RL algorithms means that the agent has to develop a *policy* for acting in an unknown environment by trial-and-error (Sutton and Barto, 1998). In every such interaction the agent performs an action on the environment and receives a *percept*. The percept consists of the current configuration of the environment, called *state*, and a scalar feedback signal, called *reward*. The reward signal indicates how good the sequence of *actions* of the agent was. The *goal* of an RL agent is based on the concept of the *reward hypothesis*:

Definition 1 (Reward Hypothesis)**.** *(Sutton, 1999) Any notion of a goal or purpose of an intelligent agent can be described as the maximization of expected cumulative reward.*

The existence of an extrinsic feedback signal makes RL algorithms also somewhat supervised in nature - thus RL algorithms are in some sense both supervised and unsupervised (Barto and Dietterich, 2004).

As an example, consider an agent playing a car racing game in which the goal is to reach the finish line as soon as possible. To model the goal as a cumulative reward maximization problem, we give the agent a negative reward every time step, thereby

incentivizing the agent to reach the finish line as quickly as possible. This example illustrates how an objective can be modelled as the maximization of expected cumulative reward, and the *goal* of an agent as a sequence of actions that achieves it. The interaction between agent and environment continues until the agent converges to an optimal sequence of actions for each state in the environment. This interaction is called the agent-environment interaction cycle, as illustrated in Figure 1.1. Each iteration of the interaction is called a time-step, often denoted by the subscript t to distinguish states, actions, and percepts between time-steps.

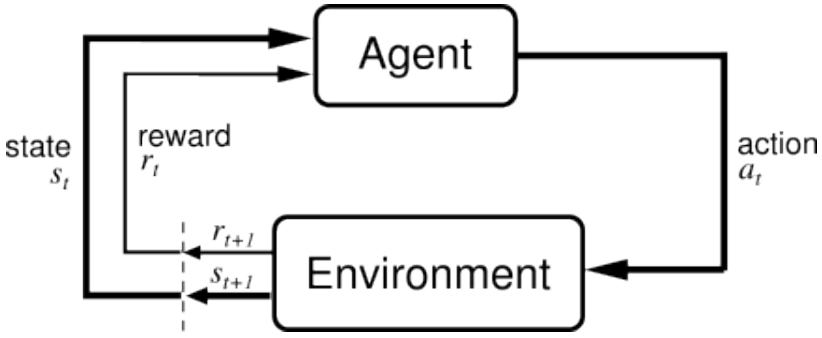

Figure 1.1: The agent-environment interaction cycle (Sutton and Barto, 1998)

1.2 The Exploration/Exploitation Dilemma

In an online decision-making setting such as the reinforcement learning problem, an agent is faced with two choices - *explore* or *exploit*. The term *exploration* in an active learning system is defined as the process of deliberately taking a non-greedy action with the sole aim of gathering more information about the environment. Exploration plays a fundamental role in reinforcement learning algorithms. It is born out of the notion that an optimal long-term policy might involve short-term sacrifices. Alternatively, *exploitation* is the act of taking the best possible action given the current information about the environment. A central challenge in reinforcement learning is to find the sweet spot between exploration and exploitation, i.e., to figure out when to explore and when to exploit. This problem is known as the *exploration-exploitation dilemma*.

At present there are a number of provably efficient exploration methods that are effective in environments with low-dimensional state-action spaces. Most of the exploration algorithms which enjoy strong theoretical guarantees implement the so-called "Optimism in the Face of Uncertainty" (OFU) principle. This heuristic encourages the agent to be optimistic about the reward it might attain in less explored parts of the environment. The agent seeks out states with higher associated uncertainty, and in doing so reduces its uncertainty in a very efficient way. Many algorithms that implement this heuristic do so by adding an exploration bonus to the agent's reward signal. This bonus is usually a function of a state visit-count; the agent receives higher exploration bonuses for exploring less frequently visited states (about which

it is less certain).

Unfortunately, these algorithms do not scale well to high-dimensional environments. In these domains, the agent can only visit a small portion of the state space while it is training. The visit-count for most states is always zero, even after training is finished. Nearly all states will be assigned the same exploration bonus throughout training. This renders the bonus useless as a tool for efficient exploration. All unvisited states appear to the agent as equally uncertain. This problem arises because these count-based OFU algorithms fail to generalise the agent's uncertainty from one context to another. Even if an unvisited state has very similar features to a frequently visited one, the agent will treat the former as a complete unknown. Consequently even the sophisticated algorithms that are suitable for the high-dimensional setting – e.g. those that use deep neural networks for policy evaluation – tend to use simple, inefficient exploration strategies.

Success in the high-dimensional setting demands that the agent represent the state space in a way that allows generalisation about uncertainty. This sort of generalisation would allow that the agent's uncertainty be lower for states with familiar features, and higher for states with novel features, even if those exact states haven't been visited. What we require, then, is an efficient method for computing a suitable similarity measure for states. That is the key challenge addressed in this thesis.

1.3 Summary of Contributions

This thesis presents a new count-based exploration algorithm that is feasible in environments with large state-action spaces. It can be combined with any value-based RL algorithm that uses linear function approximation (LFA). The principal contribution is a new method for computing generalised visit-counts. Following Bellemare et al. (2016), we construct a visit-density model in order to measure the similarity between states. Our approach departs from theirs in that we do not construct our density model over the raw state space. Instead, we exploit the feature map that is used for value function approximation, and construct a density model over the transformed feature space. This model assigns higher probability to state feature vectors that share features with visited states. Generalised visit-counts are then computed from these probabilities; states with frequently observed features are assigned higher counts. These counts serve as a measure of the uncertainty associated with a state. Exploration bonuses are then computed from these counts in order to encourage the agent to visit regions of the state-space with less familiar features.

Our density model can be trivially derived from any feature map used for LFA, regardless of the application domain, and requires little or no additional design. In contrast to existing algorithms, there is no need to perform a special dimensionality reduction of the state space in order to compute our generalised visit-counts. Our method uses the same lower-dimensional feature representation to estimate value and to estimate uncertainty. This makes it simpler to implement and less computationally expensive than some existing proposals. Our evaluation demonstrates that

this simple approach achieves near state-of-the-art performance on high-dimensional RL benchmarks.

Chapter 2

Background and Related Work

'Each night, when I go to sleep, I die.
And the next morning, when I wake up,
I am reborn.'

Mahatma Gandhi

In this chapter we give a formal treatment of the RL problem. We then provide a taxonomy of RL algorithms and the challenges posed by classical RL algorithms. Further, we talk about relevant research findings on how to solve these challenges.

2.1 Classical Reinforcement Learning

In Classical RL (CRL), the environment is assumed to be fully observable, ergodic, and every state has the *Markov property*. The branch of reinforcement learning where these assumptions are lifted is called General Reinforcement Learning (GRL) (Hutter, 2005).

Definition 2 (Markov property)**.** *Future states are only dependent on the current states and action, and are independent of the history of percepts. Formally,*

$$P(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0) = P(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t)$$

for all s', r*, and histories* $s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0$

A *Markov Decision Process* (MDP) captures the above assumptions about the environment, and so in the CRL context the environment is modelled as an MDP (Puterman, 1994). Thus, the CRL problem now reduces to the problem of finding an optimal policy for an unknown MDP.

Definition 3 (Markov Decision Process)**.** *A Markov Decision Process is a Tuple* $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ *representative of a fully-observable environment in which all states are Markov.*

- $\mathcal{S}$ *is a finite set of states*
- $\mathcal{A}$ *is a finite set of actions*

- $\mathcal{P}^a_{ss'} = \mathbb{P}[s_{t+1} = s' \mid s_t = s, a_t = a]$ *are the transition probabilities*
- $\mathcal{R}^a_{ss'} = \mathbb{E}[r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s']$ *is the expected value of the reward resulting from the transition* s, a, s'
- γ *is the discount factor which weights the relative importance of immediate rewards to future rewards.*

If the dynamics (transition and reward distributions) of the MDP are known, then we can use dynamic programming methods to directly plan on the MDP to find an optimal policy. In the RL context, in which the system dynamics are unknown, we have to use iterative RL algorithms such as TD-learning (Sutton, 1988) to find a good policy asymptotically.[1]

Definition 4 (Policy)**.** *A policy may be deterministic or stochastic. A deterministic policy is a mapping from the states to actions.*

$$\pi : \mathcal{S} \to \mathcal{A}$$

A stochastic policy is a probability distribution over the set of actions given a state.

$$\pi(a \mid s_t = s)$$

Value

The most common way to characterize the quality of a given policy is to define a function that computes how valuable it is to follow the policy from a given state (or state-action pair). This notion of value is expressed in terms of future rewards the agent could expect, if it had chosen to follow the given policy.

Definition 5 (State-Value Function)**.** *The state-value function,* $V^\pi(s)$ *is a mapping from states to* $\mathbb{R}$*. The value of a state* $s \in \mathcal{S}$ *under policy* π *is the expected discounted cumulative reward given that the agent starts in state* s *and follows policy* π *thereafter.*

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right]$$

Definition 6 (Action-Value Function)**.** *The action-value function,* $Q^\pi(s, a)$ *is a mapping from state-action pairs to* $\mathbb{R}$*. The action-value of the state-action pair* (s, a) *under policy* π *is the expected discounted cumulative reward given that the agent starts in state* s*, takes action* a*, and follows policy* π *thereafter.*

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right]$$

[1] Asymptotic analysis is one of the few theoretical tools we have to analyse RL algorithms in a domain-agnostic way.

Bellman Equations

Bellman equations form the basis for how to compute, approximate, and learn value functions in the RL setup (Sutton and Barto, 1998). They arise naturally from the structure of an MDP by capturing the recursive relationship between the value of a state and the value of its successor states. The two Bellman equations for the state-values and action-values can be defined as follows.

Definition 7 (Bellman Equation for state-value function of an MDP)**.**

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \left[\sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s') \right] \right]$$

Definition 8 (Bellman Equation for action-value function of an MDP)**.**

$$Q^{\pi}(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma \sum_{a' \in \mathcal{A}} \pi(a' \mid s') Q^{\pi}(s',a') \right]$$

We can now use the value function to define a partial ordering over policies. A policy is said to be better than another when the expected return of one policy is greater than or equal to the other for all states. Formally, $\pi \gtrsim \pi' \iff V^{\pi}(s) \geq V^{\pi'}(s) \quad \forall s \in \mathcal{S}$. From the imposed partial ordering it has been shown that there exists at least one policy, π^*, such that $\pi^* \gtrsim \pi$ for all policies π, although it might not be unique (Bertsekas and Tsitsiklis, 1996). The Bellman Optimality Equations provide a mathematical framework for talking about the optimal policy just by replacing the sum over actions with a max operator. Intuitively, this represents a policy that is greedy with respect to the value of its successor states.

Definition 9 (Bellman Optimality Equation for state-values)**.**

$$V^{\pi^*}(s) \equiv V^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma V^*(s') \right]$$

Definition 10 (Bellman Optimality Equation for action-values)**.**

$$Q^{\pi^*}(s,a) \equiv Q^*(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma \max_{a' \in \mathcal{A}} Q^*(s',a') \right]$$

For finite MDPs with known environment dynamics, the Bellman Optimality Equations have a unique solution. Unfortunately in the RL setup we deal with an unknown MDP. Thus, almost all of the RL algorithms approximate the Bellman Optimality Equations for an unknown MDP and try to iteratively find an optimal policy asymptotically.

2.1.1 Reinforcement Learning Algorithms

The fundamental difference between an RL problem and a *planning* problem is the knowledge of the environment dynamics. In a planning problem the model of the

environment is already known and the problem boils down to finding an optimal policy in the environment. In an RL problem, the agent is dropped into an unknown environment the dynamics of which is unknown. This makes reinforcement learning a hard problem. This distinction gives rise to two categories of RL algorithms, namely *model-based* and *model-free*.

The class of algorithms that learns the model of the environment, and then does planning within the learned model are called *model-based* RL algorithms. These algorithms learn the transition probabilities ($\mathcal{P}^a_{ss'}$) and reward functions ($\mathcal{R}^a_{ss'}$) of the MDP by iteratively simulating the environment and updating the simulation to better represent the true environment. This approach to solve unknown MDP's is computationally intensive, especially in large or continuous problems. Value iteration and policy iteration are two dynamic programming algorithms that have a planning-based approach to the RL problem. On the other hand, *model-free* algorithms directly learn the optimal policy using an intermediary quantity (usually the value-function).

Generalized Policy Iteration (GPI)

The overarching theme of almost all value-function based CRL algorithms is the back-and-forth between two interacting processes, *prediction* and *control*, eventually resulting in convergence. *Prediction* refers to policy-evaluation where the value-function is estimated for the current policy. *Control* on the other hand aims to find a policy that is greedy with respect to the current value-function (state-value or action-value).

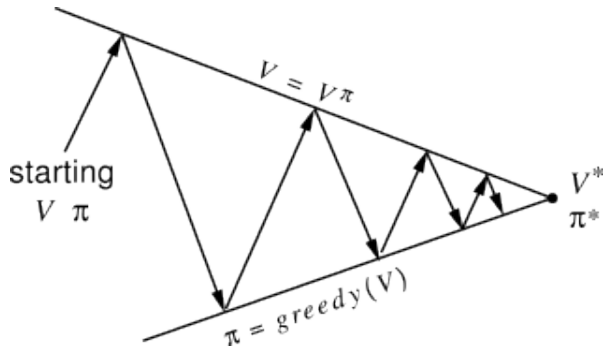

Figure 2.1: Generalized Policy Iteration (Sutton and Barto, 1998)

The 'Prediction and Control' process converges when it produces no significant change, that is, the value-function is consistent with the current policy and the policy is greedy with respect to the current value-function.

Temporal Difference Learning

Temporal Difference learning (TD learning) is a common RL algorithm; it is a model-free algorithm that combines Monte Carlo methods with the ideas from dynamic programming. TD learning allows the agent to directly learn from its experience of the environment. Following the GPI theme, we need a strategy for prediction

and control. In TD prediction we use the sampling of Monte Carlo methods and bootstrapping (updating from an existing estimate) of DP algorithms to estimate the current value-function.

Definition 11 (Update formula for state-value function)**.**

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

TD(0) is a TD learning algorithm that updates state-values after each time-step, so the learning process is fast and on-line. The target for the TD(0) update formula uses the existing estimate of $V(s_{t+1})$, hence we say the algorithm bootstraps.

As the agent interacts with the environment more, TD learning is able to generate a better estimate of the value-functions. In the limit, if each state (or state-action pair) is visited infinitely often with some additional constraints on the learning rates, convergence to the true value-function is guaranteed (Bertsekas and Tsitsiklis, 1996).

In TD control we want to optimize the value-function of an unknown environment. There are two classes of policy control methods, namely, *on-policy* and *off-policy*. On-policy control uses the policy derived from the current value-function estimate to update the future estimates. Alternatively, off-policy control uses a policy that is greedy with respect to the current value-function to estimate future value-functions. $SARSA$ (on-policy) and Q-learning (off-policy) are two popular TD control algorithms that are known to learn an MDP asymptotically (Sutton and Barto, 1998; Watkins and Dayan, 1992).

The important concept of why we are able to do model-free TD control lies in the fact that we use (state,action)-value functions instead of state-value functions.

Definition 12 (Greedy policy control)**.** *Policy improvement is done by considering a new policy, π', which is greedy with respect to the current value-function.*

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} \mathcal{R}_s^a + \mathcal{P}_{ss'}^a V(s') \qquad (\textit{Greedy w.r.t. state-value function})$$

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} Q(s, a) \qquad (\textit{Greedy w.r.t. action-value function})$$

From the above policy improvement equations we can see that in order to be greedy with respect to the state-value function, we require the model of the MDP. In contrast, if the policy is greedy with respect to the action-value function, the model dynamics of the MDP is not needed, and hence, it is *model-free*. Thus, optimizing action-value functions to learn the optimal policy is at the heart of all model-free TD control algorithms.

Challenges and Drawbacks

All the Classical Reinforcement Learning algorithms that we discussed above can be categorized as *tabular* algorithms. That is, the algorithms use a table data structure to associate each state (or state-action pair) with its current value estimate. As the agent

interacts with the environment and gains experience, the table values are updated with better estimates of its value.

The main drawback of such a method is that it scales poorly. When the state-space is very large or continuous, the fundamental requirement that the agent visits each state (or state-action pair) multiple times (or infinitely often) is not satisfied; states are at most visited once. An agent following a policy derived from these value estimates would do no better than a random policy. Moreover, the table size grows with the number of states, making storage infeasible for problems with large/continuous state-space.

A common approach to solving this problem is to find a way to *generalize* the value-function from the limited experience of the agent (Sutton and Barto, 1998). That is, we want to approximate the value-function for an unseen state (or state-action pair) from the example values it has observed so far. *Function approximation* is a generalization technique that does exactly this; it takes in observed values of a desired function and attempts to generalize an approximation of the function.

2.1.2 Function Approximation

Function approximation (FA) is an instance of supervised learning (Sutton and Barto, 1998). It is viewed as a class of techniques used to approximate functions by using example values of the desired function. In the RL context tabular methods become infeasible in large or continuous state spaces. This challenge is mitigated by employing FA techniques to predict the value-function at unseen states. However, not all FA methods are applicable to the RL setting. We require a training method which can learn efficiently from on-line, non-i.i.d. data, and also handle non-stationary target functions. The following are some of the function approximators that are used in the RL context.

- Gradient-Descent Methods
  - Artificial Neural Networks
  - Linear Combination of Features
- State Aggregation
  - k-Nearest Neighbors
  - Soft Aggregation

State aggregation is a method of generalizing function approximation in which states are grouped based on a criterion and then value is estimated as an attribute of the group. When a state is re-visited the value corresponding to the state's group gets updated.

Linear combination of features, also known as *Linear Function Approximation* (LFA), is essentially a linear mapping from the state space (of dimensionality D) to a feature space of dimension M, where often $M < D$. Each basis function of the feature

space is a mapping from the state space to a real-valued number that represents some feature of the state-space.

Definition 13 (Linear-Approximate state(action)-value function)**.** *The approximate state-value function of a state $s \in \mathcal{S}$ under a policy π is given by:*

$$\hat{V}^{\pi}(s) = \boldsymbol{\theta}^T \boldsymbol{\phi}(s) = \sum_{i=1}^{M} \theta_i \phi_i(s)$$

$$\hat{Q}^{\pi}(s,a) = \boldsymbol{\theta}^T \boldsymbol{\phi}(s,a) = \sum_{i=1}^{M} \theta_i \phi_i(s,a)$$

Where $\boldsymbol{\phi} : \mathcal{S}(\times\mathcal{A}) \to \mathbb{R}^M$, is a feature map, and $\boldsymbol{\theta} \in \mathbb{R}^M$ is the parameter vector.

LFA has sound theoretical guarantees and also is very efficient in terms of both data and computation (Sutton and Barto, 1998), making it a good candidate for the implementation of our algorithm.

As mentioned previously, FA can be regarded as a technique to develop a generalization regarding value. In order to have a good capacity to generalize, a function approximator must have relevant data about the state-space. Consider a pathological case in which the agent does not explore at all: as a result the only data available for FA would be concentrated in one region of the state space. This results in the estimation of values of unseen states being highly biased. In order to avoid this problem we have to make sure that the agent visits most regions of the state-space, that is, the agent has to explore the state-space efficiently. The main goal of this thesis is to address the problem of how to explore efficiently in large state-spaces.

2.2 Exploration Strategies for Reinforcement Learning

In Section 1.2 we described the *exploration/exploitation dilemma*, which is a fundamental problem in RL. All exploration strategies attempt to manage the trade-off between these two often opposed objectives. The simplest and most widely-used exploration strategy is known as ϵ-greedy. At each time-step t the agent chooses a greedy action with probability $1 - \epsilon$ and with ϵ probability the agent chooses a completely random action. To ensure that the policy converges to the optimal policy it has to satisfy the *GLIE* assumptions (Singh et al., 2000):

Definition 14 (Greedy in the Limit with Infinite Exploration)**.** *A policy is GLIE if it satisfies the following two assumptions.*

- *Each action is taken infinitely often in every state that is visited infinitely often,*

$$\lim_{t\to\infty} N_t(s,a) = \infty$$

Where $N_t(s,a)$ is the number of times action a has been chosen in state s up-to time-step t.

- *In the limit, the learning policy is greedy with respect to the Q-value function with probability* 1.

$$\lim_{t\to\infty} \pi_t(a \mid s) = 1, \textit{when, } a = \arg\max_{a' \in \mathcal{A}} Q_t(s, a')$$

For example, ϵ-greedy satisfies the GLIE assumptions when ϵ is annealed to zero. A common way to do this is by setting $\epsilon_t \propto 1/t$.

In small, finite MDPs ϵ-greedy satisfies the GLIE assumptions, but when the state-action space is large/continuous the first GLIE assumption is violated and hence the convergence guarantee is lost. ϵ-greedy is a naïve approach to solving the exploration problem, but we still use it in large MDPs because of its low resource requirements when compared with alternatives (Bellemare et al., 2016). In this thesis we propose a novel exploration strategy that improves upon ϵ-greedy, and provides state-of-the-art results in large problems with low computational overhead.

We now provide an exposition of various explorations strategies, their foundational principles, and an analysis of recent breakthroughs in the field of exploration.

2.2.1 Taxonomy of Exploration Strategies

The exploration-exploitation dilemma is still an open problem, but researchers have made significant inroads into understanding the nature of the problem. Sebastian Thrun classified exploration techniques into two families of exploration schemes, *directed* and *undirected* (Thrun, 1992). Undirected exploration strategies do not use any information from the environment to make an informed exploratory action; they predominantly rely on randomness to do exploration. Softmax methods and ϵ-greedy are examples of undirected exploration techniques. The *softmax* action is sampled from the *Boltzman distribution*

$$\text{Boltz}_s(a) = \frac{\exp(Q(s,a))}{\sum_{a' \in \mathcal{A}} \exp(Q(s,a'))}.$$

On the other hand, directed exploration strategies use the knowledge about the learning process to form an exploration-specific *heuristic* for action selection. This heuristic directs the agent to take those actions that maximizes the information gain about the environment. The exploration algorithm introduced in this thesis falls into the category of directed exploration algorithms. In order to put it into context, we first present an overview of the existing directed exploration strategies used in the literature.

2.3 The Optimism in the Face of Uncertainty Principle

In the following chapter we present our directed exploration method, which implements the principle of "Optimism in the Face of Uncertainty" (OFU) as a heuristic for exploration. In this section we review existing work on the OFU heuristic. The principle is succinctly captured in Osband and Van Roy (2016):

> "When at a state, the agent assigns to each action an optimistically biased while statistically plausible estimate of future value and selects the action with the greatest estimate."

OFU is a heuristic to direct exploratory actions. OFU directs the agent to take actions which have more uncertain value estimates. Instead of greedily taking the action that has the highest estimated value, that agent is encouraged to take actions which have a high *probability* of being optimal. To see that an apparently suboptimal action may indeed have a high probability of being optimal, let us take an example. Suppose that the agent has taken an action $a \in \mathcal{A}$ very often from a particular state $s \in \mathcal{S}$, and suppose that a also currently has the highest value-estimate $\hat{Q}^\pi(s,a)$ among the available actions. Now consider an alternative action $a' \in \mathcal{A}$ that has only been tried once from the state s, and suppose that the reward received was lower than $\hat{Q}^\pi(s,a)$. Action a has higher estimated value, but having tried it many times, the agent's uncertainty about its value is quite low. In contrast, the uncertainty about the value of the alternative action $\hat{Q}^\pi(s,a')$ is very high, since it has been taken so rarely. Thus, while the current estimate $\hat{Q}^\pi(s,a')$ may be lower than $\hat{Q}^\pi(s,a)$, there is a good chance that the agent was unlucky when taking a' the first time, and that the true action-value $Q^\pi(s,a')$ is much higher than both estimates. Thus it may be that a' has a higher probability of being the optimal action than does a, especially if their estimated values are quite close. The OFU heuristic would bias the agent toward taking action a' instead of the greedy action a. An agent following this heuristic will behave as if it is optimistic about action a', or more precisely, about its true action-value $Q^\pi(s,a')$. This optimism drives the agent to explore regions of the environment about which it is more uncertain.

2.3.1 OFU using Count-Based Exploration Bonuses

Most of the exploration algorithms that enjoy strong theoretical efficiency guarantees, implement the OFU heuristic. Many do so by augmenting the estimated value of a state(-action pair) with an exploration bonus that quantifies the uncertainty in that value estimate. An agent which acts greedily with respect to this augmented value function will be biased to take actions with higher associated uncertainty. Most of these algorithms are *tabular* and *count-based* in that they compute their exploration bonuses using a table of state(-action) visit-counts. The visit-count serves as an approximate measure of the uncertainty associated with a state(-action), because more novel state(-action) pairs will have lower visit-counts. State(-actions) with lower visit counts are assigned higher exploration bonuses. This drives the agent to behave optimistically and explore less frequently visited regions of the environment, which may yet prove to have higher value than familiar regions. Moreover, even if those regions turn out to yield little reward when explored, the agent will have greatly reduced its uncertainty about those regions. Indeed, the reduction in uncertainty would be much smaller if the agent were to take an action that had already been tried many times. The OFU heuristic is therefore a win-win approach for the agent. OFU algorithms are more efficient than undirected exploration strategies like ϵ-greedy because the agent

avoids actions that yield neither large rewards nor large reductions in uncertainty (Osband et al., 2016).

2.3.2 Tabular Count-based Exploration Algorithms

One of the best known OFU methods is the UCB1 bandit algorithm, which selects an action that maximises an upper confidence bound $\hat{Q}_t(a) + \sqrt{\frac{2 \log t}{N(a)}}$, where $\hat{Q}_t(a)$ is the estimated mean reward and $N(a)$ is the visit-count (Lai and Robbins, 1985). The dependence of the bonus term on the inverse square-root of the visit-count is justified using Chernoff bounds. In the MDP setting, the tabular OFU algorithm most closely resembling our method is Model-Based Interval Estimation with Exploration Bonuses (MBIE-EB) (Strehl and Littman, 2008).[2] Empirical estimates $\hat{\mathcal{P}}$ and $\hat{\mathcal{R}}$ of the transition and reward functions are maintained, and $\hat{\mathcal{R}}(s,a)$ is augmented with a bonus term $\frac{\beta}{\sqrt{N(s,a)}}$, where $N(s,a)$ is the state-action visit-count, and $\beta \in \mathbb{R}$ is a theoretically derived constant. The Bellman optimality equation for the augmented action-value function is

$$\tilde{Q}^{\pi}(s,a) = \hat{\mathcal{R}}(s,a) + \frac{\beta}{\sqrt{N(s,a)}} + \gamma \sum_{s'} \hat{\mathcal{P}}(s' \mid s,a) \max_{a' \in \mathcal{A}} \tilde{Q}^{\pi}(s',a')$$

Here the dependence of the bonus on the inverse square-root of the visit-count is provably optimal (Kolter and Ng, 2009). This equation can be solved using any MDP solution method.

While tabular OFU algorithms perform well in practice on small MDPs (Strehl and Littman, 2004), their *sample complexity* becomes prohibitive for larger problems (Bellemare et al., 2016). The sample complexity of an algorithm is a bound on the number of timesteps at which the agent is not taking an ϵ-optimal action with high probability (Kakade, 2003). Loosely speaking, it measures the amount of experience the agent must have before one can be confident it is basically performing optimally. MBIE-EB, for example, has a sample complexity bound of $\tilde{O}\left(\frac{|\mathcal{S}|^2|\mathcal{A}|}{\epsilon^3(1-\gamma)^6}\right)$. In the high-dimensional setting – where the agent cannot hope to visit every state during training – this bound offers no guarantee that the trained agent will perform well. The prohibitive complexity of these tabular OFU algorithms is due in part to the fact that a table of visit-counts is not useful if the state-action space is too large. Since the agent will only visit a small fraction of that space, the visit-count for most states will always be zero. These algorithms are therefore unable to usefully compare the novelty of two unvisited states. All unvisited states have the same visit-count, and hence the same exploration bonus. The optimistic agent will treat them all as equally novel and equally appealing.

[2] To the best of our knowledge, the first work to use exploration bonuses in the MDP setting was the Dyna-Q+ algorithm, in which the bonus is a function of the recency of visits to a state, rather than the visit-count (Sutton, 1990)

2.3.3 Generalized Visit-counts for Exploration in Large MDPs

Tabular OFU algorithms fail on high-dimensional problems because they do not allow for generalization across the state space regarding uncertainty. Every unvisited state is treated as entirely novel, regardless of any similarity between the unvisited states and the visited states in the history. In order to explore efficiently in large domains, the agent must be able to make use of the fact that some unvisited states share many features with visited states, while others share very few. If an unvisited state has almost exactly the same features as a very frequently visited one, then it should not be considered to be as uncertain as a state with unfamiliar features. An effective OFU method for these problems would not just encourage the agent to visit unvisited states, but rather would drive the agent to visit states with novel or uncommon features. We discuss this issue further in section Section 3.1.1.

Several very recent extensions of count-based exploration methods have achieved this sort of generalisation regarding uncertainty, and have produced impressive results on high-dimensional RL benchmarks. These algorithms closely resemble MBIE-EB, but they substitute the state-action visit-count for a *generalised visit-count* which quantifies the similarity of a state to previously visited states. Bellemare et al. (2016) construct a Context Tree Switching (CTS) density model over the state space such that higher probability is assigned to states that are more similar to visited states (Veness et al., 2012). A state pseudocount is then derived from this density. A subsequent extension of this work replaces the CTS density model with a neural network (Ostrovski et al., 2017). Another recent proposal uses locality sensitive hashing (LSH) to cluster similar states, and the number of visited states in a cluster serves as a generalised visit-count (Tang et al., 2016). As in the MBIE-EB algorithm, these counts are used to compute exploration bonuses. These three algorithms outperform random strategies, and are currently the leading exploration methods in large discrete domains where exploration is hard.

Before presenting our optimistic count-based exploration method in the following chapter, we now briefly canvas two alternative frameworks for directed exploration, and discuss their limitations.

2.4 Bayes-Adaptive RL

In the Bayesian approach to model-based reinforcement learning, we maintain a posterior distribution over the possible models of the environment given the experience of the agent (Dearden et al., 1998). Bayesian inference is used to update the posterior with new information as the agent interacts with the environment, and also to incorporate the agent's prior distribution over the transition models.

Since the posterior is maintained over all possible models we can now talk about the uncertainty pertaining to what is the best action to take. This uncertainty is modelled as a Markov Decision Process defined over a set of *hyper-states*. A hyper-state acts as an information state which summarizes the information accumulated so far. This augmented MDP, often referred to as the Bayes-Adaptive MDP (BAMDP),

can be solved with standard RL algorithms (Duff, 2002). In this framework an agent acting greedily in the BAMDP whilst updating the posterior acts optimally (according to its prior belief) in the original MDP. The Bayes-optimal policy for the unknown environment is the optimal policy of the BAMDP, thereby providing an elegant solution to the exploration-exploitation trade-off.

Unfortunately, the cardinality of the hyper-states grows exponentially with the planning horizon thereby rendering exact solution to the BAMDP computationally intractable for large problems (Duff, 2002).

2.5 Intrinsic Motivation

The final directed exploration heuristic that we discuss is born out of the so-called *intrinsic motivation* framework. There appears to be a growing scientific consensus in developmental psychology that human beings, from infants to adults, develop their understanding of the world using certain cognitive systems such as intuitive theories, social-structures, spatial systems, etc. (Spelke and Kinzler, 2007; Lake et al., 2016). During curiosity-driven, creative, or risk-taking activities, rational agents use this understanding to generate *intrinsic goals*. Accomplishing these intrinsic goals leads to the accumulation of *intrinsic rewards*, thereby exhibiting an innate desire to explore, manipulate, or probe their environment (Oudeyer, 2007).

Drawing parallels to reinforcement learning, the goal of a traditional RL agent is to maximize its expected cumulative reward. This behaviour is extrinsically motivated since the reward signal is external to an agent. We say that an agent is *intrinsically motivated* if it has intrinsic goals and rewards. In the context of exploration for RL, the aim of the intrinsic motivation approach is to use intrinsic reward as a *heuristic* that assigns an *exploratory value* to the agent's actions. For example, an agent may receive intrinsic rewards for visiting novel parts of the environment that need further exploration (Thrun, 1992).

Many formulations that quantify the exploratory value of an action has been put forth, and most of them augment the environment's reward function so as to motivate directed exploration. Schmidhuber (2010) proposed a measure for intrinsic motivation by taking into account the improvement a learning algorithm effected on its predictive world model. This measure tracks the progress of an agent's ability to better compress the history of states and actions (Steunebrink et al., 2013). Another framework for intrinsically motivated learning is to maximize the *mutual information*. An intrinsic reward measure called *empowerment* is formulated by searching for the maximal mutual information (Mohamed and Rezende, 2015). The notion of maximizing information gain was demonstrated in a humanoid robot by the introduction of *artificial curiosity* (Schmidhuber, 1991) as an *intrinsic goal* (Frank et al., 2014).

These formulations have some major drawbacks which hinder their suitability as exploration heuristics. Firstly, they fail to provide any strong theoretical guarantees of efficient exploration. Leike (2016) pointed out that since none of these heuristics take into account the reward structure of the problem, they do not distinguish be-

tween regions of high and low expected reward. Secondly, these algorithms require that we maintain the environment dynamics of the underlying MDP, which prevents us from easily integrating them with model-free algorithms. Another major drawback is the computational overhead associated with calculating the heuristic. For problems with large state/action spaces, computing the intrinsic reward becomes intractable for many heuristics (Bellemare et al., 2016). Most problems of interest have extremely large state spaces, and hence the intrinsic motivation heuristic is currently impractical as an exploration strategy in these domains.

Chapter 3

Exploration in Feature Space

'To wander is to be alive.'

Roman Payne, Europa

In this chapter we introduce a simple, optimistic, count-based exploration strategy that achieves state-of-the-art results on high-dimensional RL benchmarks. In Section 3.1 we begin by discussing the drawbacks of current exploration strategies. In Section 3.2 we provide an exposition of the core ideas that underpin our algorithm. Finally, in Section 3.3 we present our algorithm, as well as a number of related theoretical results.

3.1 Drawbacks of Existing Exploration Methods for Large MDPs

We introduced count-based exploration strategies for large MDPs in section Section 2.3.3. Even though they are the current state-of-the-art exploration algorithms in these domains, we consider that there are some potential drawbacks to their common approach to estimating novelty. The motivation for our algorithm arises from trying to avoid these drawbacks.

3.1.1 Choosing a Novelty Measure

The aforementioned algorithms compute a generalized visit-count. This generalized count is a novelty measure that quantifies the (dis)similarity of a state to those in the history. These algorithms drive the agent towards regions of the state space with high novelty. However, the effectiveness of these novelty measures depends on the way in which they measure the similarity between states. If this similarity measure is not chosen in a principled way, states may deemed similar in ways that are not relevant to the given problem. Let us explore this issue by taking an example.

Example 1 (Confounded novelty)**.** *Alice is a foodie. She wants to explore the myriad restaurants that are open in her city. Suppose that Alice's novelty measure treats restaurants as similar if they are geographically close. Alice consults her novelty measure to choose a restaurant she has not tried yet, and it returns a Chinese restaurant in a distant suburb*

that she has not visited before. Alice scratches her head thinking: 'I have been to a tonne of Chinese restaurants; if only my novelty measure understood that and suggested a different cuisine!' Unfortunately, her novelty measure considers this restaurant very dissimilar from the Chinese restaurants she has visited, simply because it is geographically distant from them.

The problem here is that Alice's novelty measure does not know anything about which features matter when evaluating the novelty of a restaurant. Let us now look at an example from the recent exploration literature where this problem can be clearly observed.

Inappropriate Novelty Measures in Practice

The problems that can arise from an unprincipled choice of novelty measure are well illustrated in the experimental evaluation of Stadie et al. (2015). Their algorithm uses an autoencoder to encode the state-space into a lower dimensional representation. The encoding is then fed into a model dynamics prediction neural network which estimates the novelty by providing an error-based bonus. This method, called Model Prediction Exploration Bonuses (MP-EB), uses an error based estimator and is different from the visit-density model of Bellemare et al. (2016), but they both estimate novelty. To generalize regarding value they use the DQN network, and so we refer to their algorithm as DQN+MP-EB.

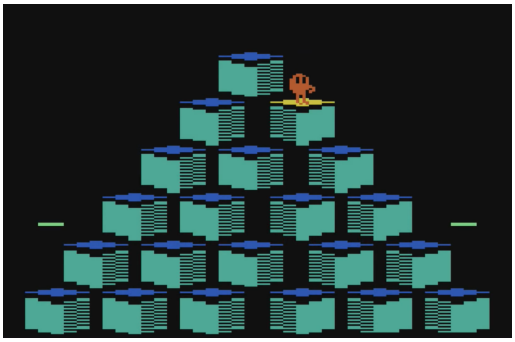

Figure 3.1: Q*bert Level 1

Figure 3.2: Q*bert Level 2

Figure 3.3: Two levels of the Atari2600 game Q*bert

During empirical evaluation of their algorithm an anomaly was detected in the game Q*bert[1] from the Arcade Learning Environment[2] (ALE) benchmarking suit. DQN+MP-EB algorithm scored lower than the baseline algorithm, DQN+ϵ-greedy. They attributed this anomaly to the fact that during each level change of Q*bert, the color of the game changes dramatically, but neither the objective nor the structure of the level changes (Figure 3.3). When their agent reached level 2 (Figure 3.2), it

[1] In Q*bert, the goal of the agent is to jump on all the cubes without falling off the edge, or being captured.

[2] ALE is a performance evaluation platform consisting of Atari2600 games. It is considered as the standard performance test bed for RL algorithms. We'll discuss in depth about ALE in Chapter 4.

perceived the state to be completely novel because MP-EB is sensitive to color. This tricked MP-EB into assigning high exploration bonus to all the states even though the action-values of the states hadn't changed. Hence the policy of the agent was impacted adversely.

The pathology of DQN+MP-EB in the Q*bert game highlights a serious problem with current novelty estimators they do not take into account the relevancy of a state to the task an agent is trying to accomplish. We argue that a measure of novelty should not just be an arbitrary generalized representation of how many times an agent has visited a state, but should ideally be a measure of dissimilarity in facets that are relevant to the agent's goal. Two states can be different in many ways; the challenge is to find out a similarity metric which is effective in achieving the agent's goal optimally. In Example 1, Alice's novelty measure did not know that suggesting a restaurant with a different cuisine would be more relevant to her task, thereby naively suggesting a geographically distant unvisited restaurant.

3.1.2 Separate Generalization Methods for Value and Uncertainty

We contend that this deficiency is not peculiar to MP-EB, but rather that it may arise whenever the novelty measure is not designed to be task-relevant. Indeed, all of the aforementioned algorithms which compute a novelty measure share a common structure which leaves them vulnerable to this problem. Each algorithm has two quite unrelated components: a value estimator (an RL algorithm which performs policy evaluation), and a novelty estimator. Each component involves an entirely separate generalization method. The value estimator makes use of a feature representation of the state space in order to generalize about value. The novelty estimator separately utilizes a different, exploration-specific state space representation to measure the similarity between states. For example, the #Exploration algorithm of Tang et al. (2016) uses the DQN algorithm for value estimation. In order to estimate novelty, however, #Exploration maps the state space into a lower-dimensional representation using locality sensitive hashing. The similarity measure induced by the choice of hash codes is unlikely to resemble that which is induced by the features learnt by DQN. The DQN-CTS-EB algorithm of Bellemare et al. (2016) has a similar structure: DQN is used to estimate value, but the CTS density model is used to estimate novelty. Again, it is not obvious that there should be much in common between the two similarity measures induced by these different state space representations. One might think that this is natural; after all, each representation is used for a different purpose. However, there are two questions we can ask here. Firstly, is there redundant computation due to performing a dimensionality reduction of the same state-space twice? If so, can we reuse the same state space representation for both value and novelty estimation? We address these questions in the following section.

Before moving on we should note that the concerns we express in this section have already been raised in the literature. In their empirical evaluation Bellemare et al. (2016) observed that their value estimator (DQN) was learning at a much slower rate than their CTS density model (their novelty measure). The authors attribute this

mismatch to the incompatibility between novelty and value estimators. They further go on to suggest that designing density models to be compatible with value function would be beneficial and a promising research direction.

The drawbacks we presented in this section suggest that there may be much room for improvement in the design of novelty estimators for exploration. In the following sections we describe our technique for estimating novelty by factoring in the insights we gained from analyzing these drawbacks. We first provide a solid footing for some of the assumptions that we made while designing the algorithm. We then go on to present our core exploration algorithm, and then combine it with a model-free RL algorithm (SARSA(λ)). In the coming chapters we present empirical evidence that our RL algorithm achieves world-leading results on the ALE benchmarking suite.

3.2 Estimating Novelty in Feature Space

3.2.1 Motivation

Which representation of the state space is appropriate for novelty estimation? Intuitively, if we use some *parameters* to determine the value of a *state*, then naturally, two such objects are considered dissimilar only if they differ in these parameters. Analogously, if the agent is using certain features to determine the value of a state, then naturally, two such states should be considered dissimilar only if they differ in those value-relevant features. This motivates us to construct a similarity measure that exploits the feature representation that is used for value function approximation. These features are explicitly designed to be relevant for estimating value. If they were not, they would not permit a good approximation to the true value function. This sets our method apart from the approaches described in Section 2.3.3, which measure novelty with respect to a separate, exploration-specific representation of the state space, one that bears no relation to the value function or the reward structure of the MDP. We argue that measuring novelty in feature space is a simpler and more principled approach, and hypothesise that more efficient exploration will result. Our proposal ensures that generalization regarding novelty is done in the same space as generalization regarding value. Figure 3.4 illustrates the basic structure of our proposed novelty estimator.

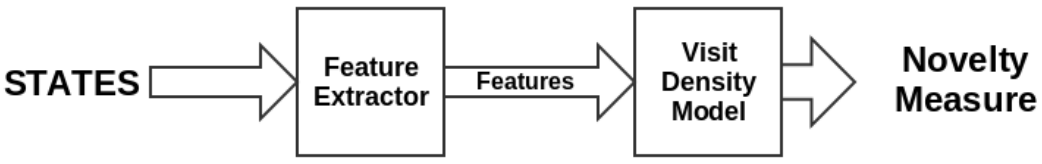

Figure 3.4: Novelty Measure in Feature Space

Let us make the idea more concrete with our running example.

Example 2 (Value-relevant exploration)**.** *After Alice's disappointing restaurant visit last time, she tweaked her novelty estimator such that it now generalizes based on value-relevant features like the type of cuisine, the star rating, and the other features that truly determine the quality of Alice's dining experience. When Alice is ready to try something new, she can rest assured that it's going to be something novel in a way that is meaningful.*

3.2.2 Design Decisions

Our exploration strategy, henceforth known as $\boldsymbol{\phi}$*-exploration bonus* ($\boldsymbol{\phi}$-EB), can be thought of as exploration in the feature space. This makes the existence of a feature map crucial to our strategy. Therefore we require that our algorithm be compatible with Linear Function Approximation (LFA). Before the advent of neural networks and subsequently DQN, large RL problems used linear function approximation to estimate the value of a state. Our decision to use LFA as our value prediction module has the following desirable benefits:

- **Domain Independence:** The visit-density models that we have seen so far (MP-EB, CTS-EB, PixelCNN, etc.) are designed to work with RGB pixel values from a video input. Though there are many domains that use video input to train the agent, there are equally many other domains that have nothing to do with a video input. For example, reinforcement learning is used in the financial sector to optimize portfolios, asset allocations, and trading systems (Moody and Saffell, 2001). Therefore developing a visit-density model that is domain independent is a key challenge. Our $\boldsymbol{\phi}$-EB method estimates the novelty using the same features that LFA uses to approximate the value function. This allows our exploration strategy to be compatible with any value-based RL algorithm that uses LFA.

- **Indirect dependence on LFA:** LFA is essentially a linear combination of features. The only requirement $\boldsymbol{\phi}$-EB has is the existence of a feature map, which is implicitly satisfied with LFA. Because of this indirect dependence on LFA, we hypothesis that it is possible to extend $\boldsymbol{\phi}$-EB to be compatible with value-networks that perform representation learning as well (e.g., DQN). Due to resource and time constraints we do not pursue empirical evidence for this claim, but rather leave this as a possible future extension of our research.

- **Single point of change:** The best way to assess the performance impact of changes to a system is to confine the change to a single module and then run performance tests. Following this principle, we know that SARSA(λ) is a value-based RL algorithm which uses LFA for value prediction and ϵ-greedy for exploration (Sutton and Barto, 1998). SARSA(λ) has been studied, perfected and validated through-out the ages. Therefore showcasing the performance gains achieved by replacing ϵ-greedy with our $\boldsymbol{\phi}$-EB exploration strategy allows for a sound empirical proof for the efficacy of our algorithm.

One drawback of using LFA for value prediction is that it requires a set of hand-crafted features. This is easily mitigated by choosing the Arcade Learning Environment (ALE) as our evaluation platform (Bellemare et al., 2013), combined with the Blob-PROST feature set (Liang et al., 2015). Using Blob-PROST as our feature set has and added advantage. Blob-PROST is designed to mimic the features learned by DQN, thus making our algorithm comparable with those using DQN for representation learning and value prediction. We'll discuss in depth about the ALE and Blob-PROST in Chapter 4.

3.3 The ϕ-EB Algorithm

The main original contribution of this work is a method for estimating novelty in feature space. The challenge is to do so without explicitly computing the distance between each new feature vector and all the feature vectors in the history. That approach quickly becomes infeasible because the cost of computing all these distances grows with the size of the history. Our method instead constructs a density model over feature space that assigns higher probability to states that share more features with more frequently observed states. In order to formally describe our method we first introduce some notation.

Notation

- $\boldsymbol{\phi} : \mathcal{S} \to \mathcal{T} \subseteq \mathbb{R}^M$, The feature map used in LFA. Maps the state space into an M-dimensional feature space, $\mathcal{T}$.
- $\boldsymbol{\phi}_t \equiv \boldsymbol{\phi}(s_t)$, Feature vector observed at time t, whose i^{th} component is denoted by $\phi_{t,i}$
- $\boldsymbol{\phi}_{1:t} \equiv (\boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_t) \in \mathcal{T}^t$, Sequence of feature vectors observed after t timesteps.
- $\boldsymbol{\phi}_{1:t}\boldsymbol{\phi} \equiv (\boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_t, \boldsymbol{\phi}) \in \mathcal{T}^{t+1}$, Sequence where $\boldsymbol{\phi}_{1:t}$ is followed by $\boldsymbol{\phi}$.
- $\mathcal{T}^*$, Set of all finite sequences of feature vectors.
- $\rho : \mathcal{T}^* \times \mathcal{T} \to [0, 1]$, The sequential density model (SDM) that maps a finite sequence of feature vectors to a probability distribution.

We will now present the key component of our algorithm that allows us to estimate novelty in feature space.

3.3.1 Feature Visit-Density

Definition 15 (Feature visit-density)**.** *The feature visit-density* $\rho_t(\boldsymbol{\phi}) \equiv \rho(\boldsymbol{\phi}\,;\, \boldsymbol{\phi}_{1:t})$ *at time* t *is a probability distribution over the feature space* $\mathcal{T}$*, representing the probability of*

observing the feature vector $\boldsymbol{\phi}$ after observing the sequence $\boldsymbol{\phi}_{1:t}$. It is modelled as a product of independent factor distributions $\rho_t^i(\phi_i)$ over individual features ϕ_i

$$\rho_t(\boldsymbol{\phi}) = \prod_{i=1}^{M} \rho_t^i(\phi_i)$$

This density model induces a similarity measure on the feature space. Loosely speaking, feature vectors that share component features are deemed similar. This enables us to use $\rho_t(\boldsymbol{\phi})$ as a novelty measure for states, because it represents the frequency with which features are observed in the history. When confronted with a new state, we are able to estimate how frequently its component features have occurred in the history. If $\boldsymbol{\phi}(s)$ has more novel component features, $\rho_t(\boldsymbol{\phi})$ will be lower. By using a density model we are therefore able to measure novelty in a way that usefully generalizes the agent's uncertainty across the state space. To illustrate this, let us consider an example.

Example 3. *Suppose we use a 3-D binary feature map and that after 3 timesteps the history of observed feature vectors is $\boldsymbol{\phi}_{1:3} = (0,1,0),(0,1,0),(0,1,0)$. Let us estimate the feature visit densities of two unobserved feature vectors $\boldsymbol{\phi'} = (1,1,0)$, and $\boldsymbol{\phi}'' = (1,0,1)$. Using the KT estimator for the factor models, we have $\rho_3(\boldsymbol{\phi'}) = \rho_3^1(1) \cdot \rho_3^2(1) \cdot \rho_3^3(0) = \frac{1}{8} \cdot \frac{7}{8} \cdot \frac{7}{8} \approx 0.1$, and $\rho_3(\boldsymbol{\phi}'') = \rho_3^1(1) \cdot \rho_3^2(0) \cdot \rho_3^3(1) = (\frac{1}{8})^3 \approx 0.002$. Note that $\rho_3(\boldsymbol{\phi'}) > \rho_3(\boldsymbol{\phi}'')$ because the component features of $\boldsymbol{\phi}'$ are more similar to those in the history. As desired, our novelty measure generalizes across the state space.*

Each factor distribution $\rho_t^i(\phi_i)$ is modelled using a count-based estimator. A naive option would be to use the empirical estimator which is the ratio of the number of times a feature has occurred to the total number of time steps. Another class of count-based estimators are the Dirichlet estimators which enjoy strong theoretical guarantees (Hutter, 2013). We use the Krichevsky-Trofimov(KT) estimator which is a Dirichlet-like estimator that is simple, easy to implement, scalable, and data efficient (Krichevsky and Trofimov, 1981). If $N_t(\phi_i)$ is the number of times the feature ϕ_i has been observed, then the KT estimator is given by:

$$\rho_t^i(\phi_i) = \frac{N_t(\phi_i) + \frac{1}{2}}{t+1}$$

Using independent factor distributions for modelling the probability of each feature component inherently assumes that the features are independently distributed. This is not always the case, especially in video-input based domains such as the ALE we have many features that are strongly correlated. This doesn't mean that we cannot use fully factorized distributions. One of the early assumptions made by Bellemare et al. (2016) about the density model is that the states are independently distributed. This allowed them to factorize the states, and model each factor using a position-dependent CTS[3] density model. Moreover, our empirical evaluations show that we

[3] A Bayesian variable-order Markov model.

achieve world leading results in hard exploration games suggesting that independent factored distributions produce good novelty measures. Thus by precedence and by empirical data the independence assumption on the features is a well-justified trade-off that makes the computation of novelty fast and data efficient.

3.3.2 The ϕ-pseudocount

Here we adopt a recently proposed method for computing generalised visit-counts from density models (Bellemare et al., 2016). By analogy with the pseudocounts presented in that work, we derive two ϕ-pseudocounts from our feature visit-density. Both variants presented generalize the same quantity, the state visitation count function $N_t(s)$. The expression given in the following definition is derived in Bellemare et al. (2016). We emphasize that our approach constitutes a departure from theirs, because while they derive pseudocounts from a *state* visit-density model, we do so using a *feature* visit-density model.

Definition 16 ($\boldsymbol{\phi}$-pseudocount)**.** *Let $\rho'_t(\boldsymbol{\phi}) \equiv \rho_t(\boldsymbol{\phi}\,;\, \boldsymbol{\phi}_{1:t}\boldsymbol{\phi})$[4] be the probability that the feature visit-density model would assign $\boldsymbol{\phi}$ if it was observed one more time. Then the $\boldsymbol{\phi}$-pseudocount for a state $s \in \mathcal{S}$ is given by:*

$$\hat{N}_t^{\phi}(s) = \frac{\rho_t(\boldsymbol{\phi}(s))(1 - \rho'_t(\boldsymbol{\phi}_t(s)))}{\rho'_t(\boldsymbol{\phi}(s)) - \rho_t(\boldsymbol{\phi}(s))}$$

3.3.3 The ϕ-Exploration Bonus algorithm (ϕ-EB)

Equipped with all the tools necessary for the construction of an exploration bonus we now proceed to define the $\boldsymbol{\phi}$-EB algorithm. We provide a high level flow-chart for the construction of the bonus in Figure 3.5, and the corresponding pseudo-code in Algorithm 1. Having defined the ϕ-pseudocount (a generalised visit-count), we follow traditional count-based exploration algorithms by computing an exploration bonus that depends on this count. The functional form of the bonus is the same as in MBIE-EB; we merely replace the empirical state-visit count with our ϕ-pseudocount.

Definition 17 ($\boldsymbol{\phi}$-exploration bonus)**.** *The exploration bonus for a state-action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ at time t is*

$$\mathcal{R}_t^{\phi}(s, a) = \frac{\beta}{\sqrt{\hat{N}_t^{\phi}(s)}}$$

where β is a hyper-parameter that controls the agents level of optimism.

Loosely speaking, the hyper-parameter β can be viewed as a knob that tunes the agent's confidence in its estimate of the true action-value function. Higher values of β makes the agent under-confident about value, and result in too much exploration. Very low β values do not encourage enough exploration because the exploration bonus is too small to dissuade the agent from acting greedily with respect to its

[4] Also called the *recoding probability*.

current value estimates. In both scenarios the final policy of the agent is affected adversely. The goal is to find a β value that gives good results across domains. We performed a coarse parameter sweep among the games in the ALE evaluation platform and concluded that $\beta = 0.05$ was the best value. Further details regarding the selection of β value is discussed in Section 5.2.1.

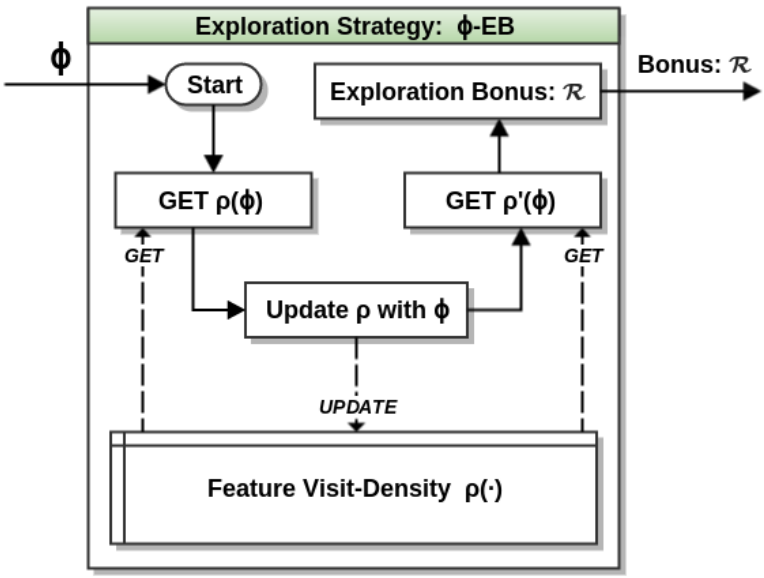

Figure 3.5: Flow Chart for computing the exploration bonus of ϕ-EB

Algorithm 1: $\boldsymbol{\phi}$-exploration bonus

Input: Density Model ρ

function FeatureVisitDensity($\boldsymbol{\phi}$)
  return $\prod_{i=1}^{M} \rho^i(\phi_i)$
end

Input: Feature Visit Count N_t; Density Model ρ; Current Timestep t

function UpdateFeatureVisitDensity($\boldsymbol{\phi}$)
  for *i=1* **to** M **do**
    $\rho^i(\phi_i) \leftarrow \frac{N_t(\phi_i) + \frac{1}{2}}{t+1}$
  end
end

function PseudoCount(p, p')
  return $\frac{p(1-p)}{p'-p}$
end

Input: LFA Feature Map $\boldsymbol{\phi}$; Exploration Coefficient β

function ExplorationBonus(s)
  $\rho(\boldsymbol{\phi}) \leftarrow$ FeatureVisitDensity($\boldsymbol{\phi}(s)$)
  UpdateFeatureVisitDensity($\boldsymbol{\phi}(s)$)
  $\rho'(\boldsymbol{\phi}) \leftarrow$ FeatureVisitDensity($\boldsymbol{\phi}(s)$)
  $\hat{N}^{\phi}(s) \leftarrow$ PseudoCount($\rho(\boldsymbol{\phi}), \rho'(\boldsymbol{\phi})$)
  return $\frac{\beta}{\sqrt{\hat{N}^{\phi}(s)}}$
end

3.3.4 LFA with $\boldsymbol{\phi}$-EB

One the advantages that we have in developing our algorithm for use with LFA is that our exploration strategy is compatible with all value-based RL algorithms that use LFA. As we will see, our empirical performance across a range of environments suggests that one can plug our exploration strategy with little to no modification into any of these algorithms and expect considerable gains in exploration efficiency. In our empirical evaluation we use SARSA(λ) with replacing traces as our value-based reinforcement learning algorithm. Algorithm 2 presents the pseudo-code for a generic RL algorithm that uses the augmented reward r^+ for updating the function

parameters $\boldsymbol{\theta}$ of the approximate action-value function $\hat{Q}(s,a) = \boldsymbol{\theta}^T \boldsymbol{\phi}(s,a)$.

Algorithm 2: LFA with $\boldsymbol{\phi}$-EB

Input: LFA Feature Map $\boldsymbol{\phi}$; Training Horizon t_{end}

$t \leftarrow 0$
Initialize arbitrary $\boldsymbol{\theta}_t$
$s_t, a_t \leftarrow$ initial state, action
while $t < t_{end}$ **do**
  $r_{t+1}, s_{t+1} \leftarrow \textsc{Act}(a_t)$
  $\mathcal{R}_t^{\phi}(s_t, a_t) \leftarrow \textsc{ExplorationBonus}(s_t)$
  $r_{t+1}^{+} \leftarrow r_{t+1} + \mathcal{R}_t^{\phi}(s_t, a_t)$
  $a_{t+1} \leftarrow \textsc{NextAction}(s_{t+1}, \boldsymbol{\theta}_t)$
  $\boldsymbol{\theta}_{t+1} \leftarrow \textsc{UpdateTheta}(r_{t+1}^{+}, \boldsymbol{\phi})$
  $t \leftarrow t + 1$
end
return $\boldsymbol{\theta}_{t_{end}}$

The functions NextAction and UpdateTheta are specific to the underlying value-based RL algorithm used, hence left unspecified. $\textsc{Act}(a_t)$ performs action a_t in the environment.

3.3.5 Complexity Analysis

Time Complexity

From Algorithm 1 it is trivial to see that a call to ExplorationBonus has a worst-case time complexity of $O(M)$, where M is the dimension of the feature space. This suggests that the time needed to compute the novelty of a state is independent of the dimension of the state-space. Also, more often than not, the dimension of the feature space is far smaller than that of the state space. Therefore, our algorithms generates significant savings in computation over other density models whose time-complexity scales with the number states. In practice, for a binary feature set like Blob-PROST we process only those features that have been observed before. This is achieved by maintaining a single prototypical factor density estimator for all previously unseen features. We'll discuss the implementation specific details in depth in Chapter 4.

Space Complexity

We look at Algorithm 2 to analyze what objects are needed to be persisted across iterations so as to facilitate calculation of the exploration bonus. Clearly the factor density estimators $\rho^i(\phi_i)$, and the feature visit count $N_t(\phi_i)$ are needed to evaluate and update the feature visit density. Therefore it can be seen that our algorithm has a worst case space complexity of $O(M)$. Again, because the features in Blob-PROST are binary valued, the KT estimator can be defined recursively. This allows for updating the factor density online without the need to maintain a feature visit count $N_t(\phi_i)$. We'll discuss more on this in Chapter 4.

3.4 Summary

In this chapter we have presented the main contribution of our research. Motivated by the drawbacks of current state-of-the-art exploration algorithms, we introduced our novel exploration algorithm called ϕ-EB. Later, we provided an exposition on the various components of the algorithm and also analysed its time and space complexity.

Now that we have presented our algorithm, we move on to implementation aspects. The next chapter provides a detailed overview of the evaluation test-bed, the software architecture, and the implementation challenges faced during the Research & Development of the algorithm.

Chapter 4

Implementation

'Any A.I. smart enough to pass a Turing test is smart enough to know to fail it.'

Ian McDonald, *River of Gods*

This chapter is dedicated to developing a technically correct implementation of our exploration strategy ϕ-EB, and its surrounding infrastructure. This allows us to perform a sound empirical evaluation which is the focus of the next chapter.

In Section 4.1 we present the high-level architecture of the whole system, and how the various components interact with each other. Later, in Section 4.2, we present the implementation of our exploration strategy, ϕ-EB. Throughout the section we also talk about the design aspects, and optimization's that went into implementing ϕ-EB.

4.1 Software Architecture

Our implementation goal is to develop an RL software agent that uses ϕ-EB as its exploration strategy. We present the high-level design of the algorithm in Figure 4.1. The presented diagram is analogous to the Agent-Environment interaction cycle (Figure 1.1), but with more granularity. From an exploration-centric standpoint, we first provide a concise overview of the components presented in the architecture, and then an exposition on the implementation details for ϕ-EB.

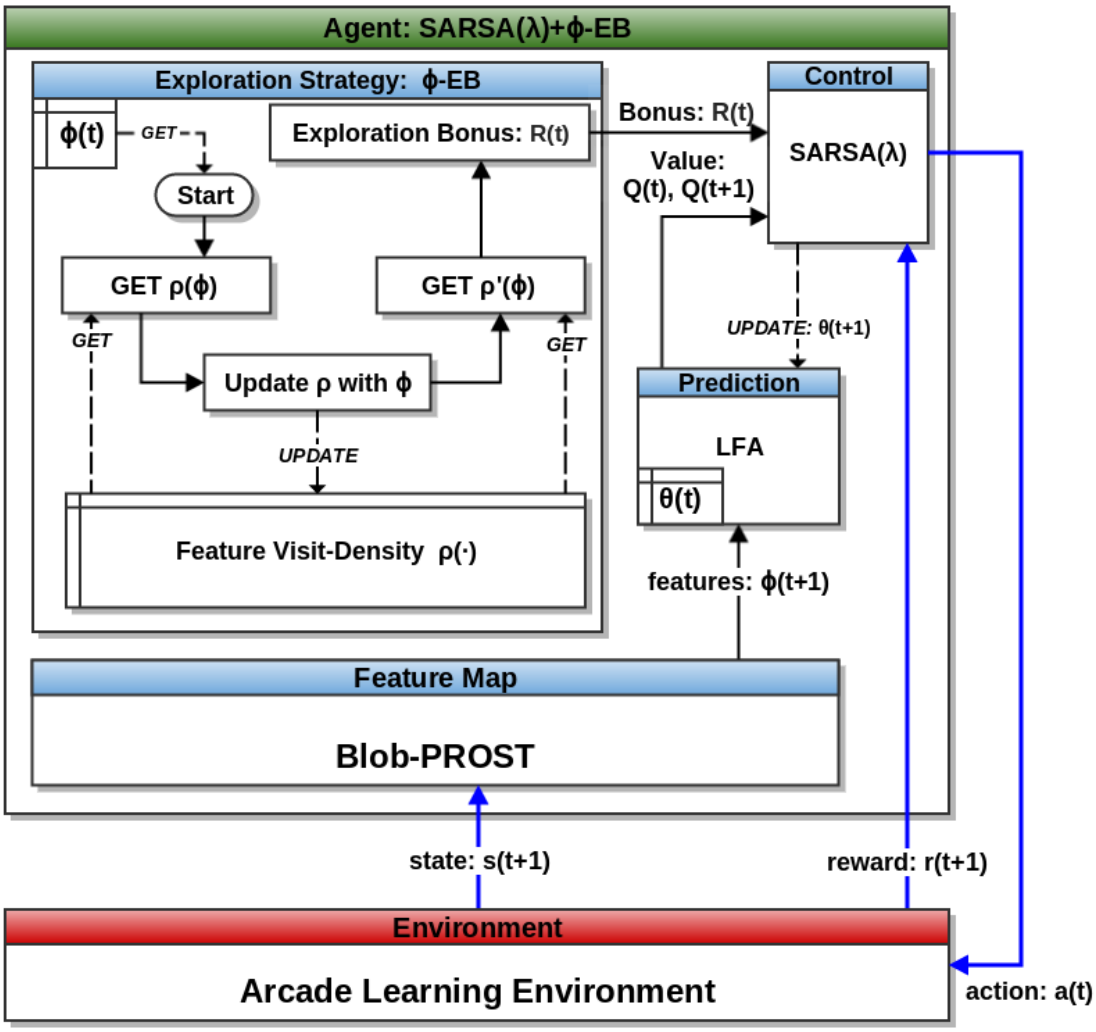

Figure 4.1: Agent-Environment interaction framework for SARSA(λ)+$\boldsymbol{\phi}$-EB.[1]

4.1.1 Modular Overview

Control

We use SARSA(λ) with replacing traces (Sutton and Barto, 1998) as our learning algorithm. This decision was driven primarily by two factors. First, using the Blob-PROST[2] feature set meant that we are locked into the framework provided by Liang et al. (2015). In our case this is in fact desirable. Replacing the exploration module of

[1] Boxes with a tiny row and column, on top and left edges respectively, denote objects stored in RAM. They can persists across cycles and episodes. Dotted arrows with instruction on them denote operation on such objects.

[2] Discussed in Section 5.1.2.

an open source, peer-reviewed and published implementation with our own exploration module enhances the credibility of any performance gains that result. Second, we need a learning algorithm that works well with Linear Function Approximation (LFA). When coupled with LFA, SARSA(λ) has better convergence guarantees than Q-learning (Melo et al., 2008). Hence, SARSA(λ) is a suitable value estimation algorithm for our agent.

Exploration Strategy

This component is our ϕ-EB exploration strategy that was proposed in Chapter 3. We implement it using the `C++11`[3] programming language. `C++` offers a significant edge over other languages in terms of efficiency and greater control over memory management. Due to the high dimensional nature of our problem, we need to extract as much performance as possible from our code. Therefore implementing the exploration strategy in `C++` is critical to the empirical success of our algorithm. Moreover, a lock-in with the framework provided by Liang et al. (2015) meant that there was no compelling reason to choose a different programming language. In the coming sections we provide a detailed look at the design and implementation of ϕ-EB.

Prediction

We use LFA to generalize the action-value function for unknown state-action pairs. For further discussion on LFA we refer the reader to Section 2.1.2. LFA uses the Blob-PROST feature set from Liang et al. (2015) to approximate the action-value function.

Feature Map

We consider the feature map to be an integral part of the agent. The ability to discern different features of a state is imperative to generalization regarding value, and by extension, to exploration. Our agent explores in the feature space, and we want to use LFA for value prediction. This necessitated the need for an efficient and effective feature set. The Blob-PROST feature set from Liang et al. (2015) is the best feature set available to date for the Arcade Learning Environment (ALE) evaluation platform. More details on Blob-PROST in Section 5.1.2.

Environment

We chose the Arcade Learning Environment as our evaluation platform for the following reasons.

- ALE contains many games which vary in degree of exploration hardness. This allows us to test the efficacy of our algorithm on a broad spectrum of games (Bellemare et al., 2016).

[3] `C++11` is a major revision of `C++`. This particular version was chosen because it makes several useful additions to core language libraries.

- ALE is widely accepted as the standard for testing RL algorithms. The vast majority of exploration specific research that is published post Bellemare et al. (2013) has adopted the ALE platform to report empirical results (Mnih et al., 2013, 2015; Stadie et al., 2015; Osband et al., 2016; Bellemare et al., 2016; Tang et al., 2016; Ostrovski et al., 2017). Therefore, in order to compare and contrast our results with existing research, it is crucial that we choose ALE as our evaluation platform.

More details on the Arcade Learning Environment in Section 5.1.1.

4.1.2 Agent-Environment Work-flow

We want to seamlessly integrate $\boldsymbol{\phi}$-EB into the control module. Therefore understanding the nuances of what happens in an agents cycle from an implementation perspective is critical. Figure 4.1 also doubles as a work-flow diagram for our agent SARSA(λ)+$\boldsymbol{\phi}$-EB. Following the usual agent-environment interaction process at timestep t, the agent performs an action a_t on the environment and receives an extrinsic reward r_{t+1}. The agent also observes the new state of the environment, s_{t+1}. Inside the agent, the Blob-PROST feature map consumes the current state s_{t+1} and returns a feature vector $\boldsymbol{\phi}_{t+1}$. The feature vector is then used by the LFA module to do value prediction. The $\boldsymbol{\phi}$-EB module uses the stored feature $\boldsymbol{\phi}_t$ to generate the exploration bonus $\mathcal{R}_t(s_t, a_t)$[4]. SARSA(λ) updates the parameters of LFA with the TD update and chooses the next action optimistically.

$\boldsymbol{\phi}$-EB

In the exploration strategy module the feature visit-density ρ is a hash map that persists in memory across cycles and episodes. Each entry of ρ is a key-value pair mapping individual features ϕ_i to its corresponding factor distribution ρ_i. In the $\boldsymbol{\phi}$-EB module shown in Figure 4.1, the flow of control is as follows: compute $\rho(\boldsymbol{\phi})$ as product of factors, update ρ with the observation $\boldsymbol{\phi}$, then compute $\rho(\boldsymbol{\phi})$ again. Now calculate the pseudo-count and subsequently the exploration bonus $\mathcal{R}_t(s_t)$. The bonus $\mathcal{R}_t(s_t)$ is considered as an intrinsic reward and is sent to the control module, SARSA(λ).

LFA

The prediction module (LFA) approximates the next state action-value function using the parameter vector $\boldsymbol{\theta}_t$ as $\hat{Q}^{\pi}(s_{t+1}, a_{t+1}) = \boldsymbol{\theta}_t^T \boldsymbol{\phi}(s_{t+1}, a_{t+1})$. LFA sends the next state Q-value to the control module, SARSA(λ). The parameter vector $\boldsymbol{\theta}_t$ is also an object that is saved in memory and persisted across cycles and episodes.

[4]Here we can see that generalization regarding value and novelty are being done in the same space.

SARSA(λ)[5]

All the results from the various modules flow into the control module SARSA(λ). The control module essentially has two tasks.

- **Choose the next action** a_{t+1}
  The next state is chosen by being greedy with respect to next state action-value obtained from LFA.
  $$a_{t+1} = \arg\max_{a \in \mathcal{A}} \left[\hat{Q}^{\pi}(s_{t+1}, a) \right]$$

- **Update $\boldsymbol{\theta}$ of LFA**
  First we augment the extrinsic reward r_{t+1} with the intrinsic reward $\mathcal{R}_t(s_t)$ obtained from $\boldsymbol{\phi}$-EB module.
  $$r^{+}_{t+1} = r_{t+1} + \mathcal{R}_t(s_t)$$
  The augmented reward r^{+}_{t+1}, the next state action-value $\hat{Q}^{\pi}(s_{t+1}, a_{t+1})$, and the current state action-value $\hat{Q}^{\pi}(s_t, a_t)$, both from LFA, is used to calculate the TD error.
  $$\delta_{t+1} = r^{+}_{t+1} + \gamma \hat{Q}^{\pi}(s_{t+1}, a_{t+1}) - \hat{Q}^{\pi}(s_t, a_t)$$
  Where γ is the discount factor. Next we update $\boldsymbol{\theta}$, and is updated using the usual TD update formula.
  $$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha \delta_{t+1}$$
  Where α is the learning rate.

Now that we have a clear idea about the surrounding infrastructure, let's move on to the implementation details of $\boldsymbol{\phi}$-EB

4.2 Implementation Details

4.2.1 Feature Visit-Density

The central data structure that stores the factor distribution of each individual feature is an `unordered_map`[6] called `fvd_map`$\langle \phi_i, \rho^i \rangle$. Each entry is a key-value pair mapping individual features to its corresponding factor distribution. This allows us to have constant time look-up for the factor distribution of any feature. At first glance of the theoretical formulation of feature visit-density (Section 3.3.1, Algorithm 1), the implementation looks straight forward. Unfortunately that is not the case. We need to take into account certain implementation specific aspects that are often subsumed by mathematical formulation. Following are some of the important implementation details that need to be considered for computing feature visit-density.

[5] For brevity we have left out the discussion on eligibility traces.
[6] Essentially a hash map

- **Sparse Feature Vector**
  In practice the feature vector $\boldsymbol{\phi}$ is the list of features that are active in the current timestep. Most of the time the set of observed features is in a vastly smaller subspace of the feature space $\mathbb{R}^M$. Therefore, iterating till M to compute the product of the factor distributions is quite wasteful. In order to overcome this we maintain a *prototype*[7] function that computes the KT-estimate of observing the feature give that it has never been observed in t timesteps. Now whenever a new feature is observed it is added to `fvd_map` with the current value of the prototype. If M_t is the total number of features observed till timestep t, then we can compute the feature visit-density in $O(M_t)$ time.

- **Numerical Stability**
  Experience has taught us that when dealing with probabilities, innocent looking formulas such as ours can be deceiving. Since we are taking product of probabilities, they are bound to numerically underflow. In our implementation, rather than computing $\prod_{i=1}^{M} \rho^i(\phi_i)$ we compute $\sum_{i=1}^{M} \log\left(\rho^i(\phi_i)\right)$. This allows us to safely perform probability calculations without the worry of underflow.

- **Inactive Features**
  During the evaluation of the feature visit-density we need to consider the factor distributions for the features that are inactive but previously observed. Since we have already observed $\boldsymbol{\phi}_t$, we can identify the features in `fvd_map` that are not active. The probability density stored in `fvd_map` against some feature ϕ_i, is the probability of ϕ_i being active. Assuming $\phi_i \notin \boldsymbol{\phi}_t$, the probability of ϕ_i not being active is given by $\left(1-\texttt{fvd\_map}[\phi_i]\right)$. Therefore when evaluating feature visit-density for $\boldsymbol{\phi}_t$ we should also factor in the probability of inactive features not occurring.

Algorithm 3 presents the implementation for computing the feature visit-density with all the above mentioned optimization/requirements. One key observation is that we return the log-probability. This is done to facilitate further log based proba-

[7] In this context, a prototype function creates an object of a specified type. Here, a KT estimator which has seen t zeros.

bility computation that occur in other modules.

Algorithm 3: Implementation of Feature Visit Density

```
   Input: Current Timestep t
1  function KT_Prototype()
2      return 0.5 / (t + 1)
3  end

   Input: Factor Distribution Map fvd_map⟨φ_i, ρ^i⟩
1  function LogFeatureVisitDensity(φ)
2      sum_log_rho ← 0
3      for i = 1 to |φ| do
4          if φ_i ∉ fvd_map.keys then                    // O(1) look-up
5              fvd_map[φ_i] = KT_Prototype()
6          end
7          sum_log_rho ← sum_log_rho + log(fvd_map[φ_i])
8      end
       /* Inactive features                                          */
9      for i = 1 to size(fvd_map.keys) do
10         if φ_i ∉ φ then                  // O(1) look-up with flag trick
11             sum_log_rho ← sum_log_rho + log(1 − fvd_map[φ_i])
12         end
13     end
14     return sum_log_rho
15 end
```

4.2.2 Updating Factor Densities

Recall that we use the Krichevsky-Trofimov (KT) estimator to compute the factor densities. Given a sequence of symbols, the KT-estimator computes the probability of the next symbol. For a binary symbol-set, the KT-estimator is given by.

$$Pr(x_{t+1} = 1 \mid x_{1:t}) = \frac{n_1 + \frac{1}{2}}{n_0 + n_1 + 1}$$

Where n_1 is the number of 1's seen so far in the sequence, and n_0 is the number of 0's seen so far.

The Blob-PROST feature set is binary valued, making the use of KT-estimators ideal. Therefore, our factor density for a feature ϕ_i being active is given by.

$$\rho^i(\phi_i) \equiv Pr(\phi_i = 1 \mid \phi^i_{1:t}) = \frac{N_t(\phi_i) + \frac{1}{2}}{t + 1}$$

And the probability for the feature being inactive is.

$$\rho^i(\phi_i = 0) = 1 - \rho^i(\phi_i = 1)$$

Where $N_t(\phi_i)$ is the number of times feature ϕ_i has been seen, and $\phi^i_{1:t}$ is the complete sequence of past observations for feature ϕ_i.

The factor density equation is neat and simple, but it requires that we maintain a count for each feature. This is an unnecessary overhead and we can do better. We now propose an update formula for $\rho^i(\phi_i)$ and derive it.

Proposition 1 (Update formula for KT-estimate $\rho^i(\phi_i)$)**.** *The factor distribution ρ^i_t at timestep t for feature ϕ_i can be updated using the following update formula.*

$$\rho^i_{t+1}(\phi_i) = \rho^i_t(\phi_i)\left(\frac{t+1}{t+2}\right) + \frac{\phi_i}{t+2}$$

Where $\phi_i \in \{0, 1\}$

Proof. From the equation for KT-estimates of $\rho^i_t(\phi_i)$ we have,

$$\begin{aligned} \rho^i_t(\phi_i) &= \frac{N_t(\phi_i) + \frac{1}{2}}{t+1} \\ \rho^i_t(\phi_i)\left(\frac{t+1}{t+2}\right) &= \frac{N_t(\phi_i) + \frac{1}{2}}{t+2} \end{aligned} \tag{1}$$

In the next timestep $t+1$, depending on the value of ϕ_i we have two cases.

- **Case 1:** Feature ϕ_i is active, i.e., $\phi_i = 1$
  The KT-estimate $\rho^i_{t+1}(\phi_i)$ can be written as,

$$\begin{aligned} \rho^i_{t+1}(\phi_i) &= \frac{N_{t+1}(\phi_i) + \frac{1}{2}}{(t+1)+1} \\ &= \frac{N_t(\phi_i) + 1 + \frac{1}{2}}{t+2} && \text{(Since } \phi_i = 1\text{)} \\ &= \frac{\left(N_t(\phi_i) + \frac{1}{2}\right) + 1}{t+2} \\ \rho^i_{t+1}(\phi_i) &= \frac{N_t(\phi_i) + \frac{1}{2}}{t+2} + \frac{1}{t+2} \end{aligned} \tag{2}$$

- **Case 2:** $\phi_i = 0$

The KT-estimate $\rho^i_{t+1}(\phi_i)$ can be written as,

$$\begin{aligned}
\rho^i_{t+1}(\phi_i) &= \frac{N_{t+1}(\phi_i) + \frac{1}{2}}{(t+1)+1} \\
&= \frac{N_t(\phi_i) + 0 + \frac{1}{2}}{t+2} && \text{(Since } \phi_i = 0\text{)} \\
&= \frac{\left(N_t(\phi_i) + \frac{1}{2}\right) + 0}{t+2} \\
\rho^i_{t+1}(\phi_i) &= \frac{N_t(\phi_i) + \frac{1}{2}}{t+2} + \frac{0}{t+2} && (3)
\end{aligned}$$

In both cases, from Eq. (2) and (3) we can see that the value ϕ_i decides the existence of an additional term. Therefore by observation we can combine the two cases as follows.

$$\rho^i_{t+1}(\phi_i) = \frac{N_t(\phi_i) + \frac{1}{2}}{t+2} + \frac{\phi_i}{t+2} \tag{4}$$

Therefore, from Eq. (1) and (4) we get,

$$\rho^i_{t+1}(\phi_i) = \rho^i_t(\phi_i)\left(\frac{t+1}{t+2}\right) + \frac{\phi_i}{t+2}$$

□

Algorithm 4 presents the algorithm for updating the factor distributions. It uses the update formula presented in Proposition 1 to efficiently update the factor distributions. In the implementation we can see that the update is performed in a two part manner with linear time complexity, rather than a naive double-loop search.

Algorithm 4: Factor Distribution Update

Input: Factor Distribution Map `fvd_map`$\langle \phi_i, \rho^i \rangle$; Current Timestep t

function UPDATE($\boldsymbol{\phi}$)
  for $i = 1$ **to** *`size(fvd_map.keys)`* **do**
    `fvd_map`$[\phi_i]$ = `fvd_map`$[\phi_i] \cdot \left(\frac{t+1}{t+2}\right)$
  end
  for $i = 1$ **to** $|\boldsymbol{\phi}|$ **do**
    `fvd_map`$[\phi_i]$ = `fvd_map`$[\phi_i] + \frac{1}{t+2}$
  end
end

4.2.3 Exploration Bonus

This is the entry point for our exploration strategy $\boldsymbol{\phi}$-EB. Due to the modular design of our algorithm, this function mostly acts like a hub that calls other functions sequentially to get the data required to calculate the exploration bonus. Algorithm 5 presents the implementation to calculate exploration bonus. Note that the probabilities are in log space to avoid numerical stability issues.

Algorithm 5: Exploration Bonus

Input: Exploration Coefficient β

function ExplorationBonus($\boldsymbol{\phi}$)
- $\log(\rho) \leftarrow$ LogFeatureVisitDensity($\boldsymbol{\phi}$)
- Update($\boldsymbol{\phi}$)
- $\log(\rho') \leftarrow$ LogFeatureVisitDensity($\boldsymbol{\phi}$)
- $\hat{N} \leftarrow \frac{1}{e^{\left(\log(\rho') - \log(\rho)\right)} - 1}$ `// Pseudo-count`
- **return** $\frac{\beta}{\sqrt{\hat{N}}}$ `// Exploration Bonus`

end

4.2.4 Action Selection

In the early stages of the project, our agent was facing some inexplicable issues. It had really slow learning progress, and was getting stuck with a single action for long periods of time. Fortunately, we had rich logs that helped us in identifying a pattern to the problem.

We observed that during the initial training cycles, the value predictions from LFA had very high variance due to lack of enough samples. In cases when there was an abnormally high Q-value, our greedy optimistic agent always kept taking the same action over and over again in a loop. We initially thought that, decay in the corresponding exploration bonus, coupled with increase in optimistic estimates for other states would lead to the agent breaking out of the loop. Even though eventually the agent got out of the loop, it happened only after an exorbitantly large number of episodes. From the logs we observed that each TD update only effected a small change, and hence the reason why it took a large number of episodes to overcome abnormally high Q-value.

If we were using ϵ-greedy as the exploration strategy this would not be a problem. With ϵ-greedy, the agent takes more exploratory action in the initial training cycles. Even if LFA produces highly varying Q-values initially, the agent doesn't get stuck for more that a few cycles. Thus, it can be noted that random exploration at the beginning helps stabilize the action-values predicted by LFA.

Our goal is to replace ϵ-greedy with our intrinsically motivated exploration strategy $\boldsymbol{\phi}$-EB. Unfortunately, the removal of ϵ-greedy meant that the agent's policy is now deterministic and has the above debilitating side-effect. In order to solve this

crippling issue we experimented two approaches.

- **Combine ϕ-EB with ϵ-greedy**
  A similar problem was reported by Bellemare et al. (2016). Their solution was to use ϵ-greedy, not as an exploration strategy, but as a tool to introduce stochasticity in the agents policy. During the initial training cycles, when there is high variance from the LFA estimates, taking a purely random action allows the agent to get out of the greedy action loop. In this experiment ϵ-greedy is implemented in the usual way - with probability ϵ take a random action, and a greedy optimistic action otherwise. Algorithm 6 presents the implementation.

- **Combine ϕ-EB with Boltzmann distributed action selection**
  One motivation for our research is to make sure that the agent does not take purely random actions. The approach from Bellemare et al. (2016) described above introduces purely random actions. We present an alternate approach which introduces stochasticity but in a directed manner.

  We split our optimistic Q functions into two functions, $Q^{\mathcal{E}}$ and $Q^{\mathcal{I}}$. $Q^{\mathcal{E}}$ is trained using the extrinsic reward, whereas $Q^{\mathcal{I}}$ is trained on the exploration bonus from ϕ-EB[8]. The motivation here is that we now have a value-function $Q^{\mathcal{I}}$ that directs the exploratory actions of the agent. For action selection we construct the optimistic value function as the summed value function $Q = Q^{\mathcal{E}} + Q^{\mathcal{I}}$. During action selection, with probability $(1 - \epsilon)$ the agent takes the action that is greedy with respect to Q, otherwise the agent takes a Boltzmann distributed random action. The Boltzmann distribution is constructed from the $Q^{\mathcal{I}}$ values using the `discrete_distribution` standard library. Hence the selected random action is more likely to be an action that has higher exploratory value. Algorithm 7 presents implementation for this approach.

Theoretically, the only difference between the above two approaches is the action selection process during exploration. The first approach takes a uniformly random action, whereas the second one takes a Boltzmann-distributed random action. Therefore during the implementation of the learning algorithm we implement the second approach, and swap the action selection process with the first for experimentation.

[8]When using LFA, training is done on the LFA parameters. Therefore we essentially maintain two sets of parameters, $\boldsymbol{\theta}^{\mathcal{E}}$ and $\boldsymbol{\theta}^{\mathcal{I}}$

Algorithm 6: Action Selection: ϵ-Greedy

```
1 function NextAction(Q)
    /* Q contains Q-values ∀a ∈ A for some state.                    */
2   a = arg max_{a∈A} Q(a)
    /* rand(0,1) generates random number between 0,1                 */
3   if rand(0,1) < ε then
      /* randInt(1,x) generates a uniformly random integer
         between 1,x                                                 */
4     i = randInt(1,|A|)
5     return a_i                    // random action
6   end
7   return a                  // Greedy Optimistic action
8 end
```

Algorithm 7: Action Selection: Boltzmann Distributed

```
1 function NextActionBoltz(Q^E, Q^I)
    /* Q^E, Q^I contains Q-values ∀a ∈ A for some state.             */
2   a = arg max_{a∈A} {Q^E(a) + Q^I(a)}
    /* rand(0,1) generates random number between 0,1                 */
3   if rand(0,1) < ε then
      /* boltzDistInt(W,1,x) generates an integer between 1,x
         that is Boltzmann distributed according to W               */
4     i = boltzDistInt(Q^I, 1, |A|)
5     return a_i        // Boltzmann distributed random action
6   end
7   return a                  // Greedy Optimistic action
8 end
```

4.2.5 SARSA(λ)+$\boldsymbol{\phi}$-EB

Now that we have all the modules necessary for learning, we present the implementation for our agent in Algorithm 8.

Algorithm 8: Reinforcement Learning with SARSA(λ) and $\boldsymbol{\phi}$-EB exploration

Input: Feature Map $\boldsymbol{\phi} : \mathcal{S} \to \mathbb{R}^M$; Training Horizon t_{end}

1 $t \leftarrow 0$

`/* Each θ are an` $|\mathcal{A}| \times M_t$ `matrix. */`

2 Initialize arbitrary $\boldsymbol{\theta}_t^{\mathcal{E}}, \boldsymbol{\theta}_t^{\mathcal{I}}$ $s_t \leftarrow$ Initial state

3 $Q_t^{\mathcal{E}} \leftarrow \boldsymbol{\theta}_t^{\mathcal{E}} \boldsymbol{\phi}(s_t)$ `// Vector containing` $Q^{\mathcal{E}}$`-values` $\forall a \in \mathcal{A}$

4 $Q_t^{\mathcal{I}} \leftarrow \boldsymbol{\theta}_t^{\mathcal{I}} \boldsymbol{\phi}(s_t)$ `// Vector containing` $Q^{\mathcal{I}}$`-values` $\forall a \in \mathcal{A}$

5 $a_t \leftarrow \textsc{NextActionBoltz}(Q_t^{\mathcal{E}}, Q_t^{\mathcal{I}}, s_t)$

6 **while** $t < t_{end}$ **do**

`/* Re-estimate Q-values with updated θ values. */`

7 $Q_t^{\mathcal{E}} \leftarrow \boldsymbol{\theta}_t^{\mathcal{E}} \boldsymbol{\phi}(s_t)$

8 $Q_t^{\mathcal{I}} \leftarrow \boldsymbol{\theta}_t^{\mathcal{I}} \boldsymbol{\phi}(s_t)$

9 $r_{t+1}, s_{t+1} \leftarrow \textsc{Act}(a_t)$ `// Perform action in ALE.`

10 $\mathcal{R}_t^{\phi} \leftarrow \textsc{ExplorationBonus}(s_t)$`// Compute Intrinsic reward.`

`/* Predict next state Q-values. */`

11 $Q_{t+1}^{\mathcal{E}} \leftarrow \boldsymbol{\theta}_t^{\mathcal{E}} \boldsymbol{\phi}(s_{t+1})$

12 $Q_{t+1}^{\mathcal{I}} \leftarrow \boldsymbol{\theta}_t^{\mathcal{I}} \boldsymbol{\phi}(s_{t+1})$

`/* Boltzmann distributed action selection. */`

13 $a_{t+1} \leftarrow \textsc{NextActionBoltz}(Q_{t+1}^{\mathcal{E}}, Q_{t+1}^{\mathcal{I}})$

`/* Alternatively: ε-greedy action selection. */`

`//` $a_{t+1} \leftarrow$ `NextAction`$(Q_{t+1}^{\mathcal{E}} + Q_{t+1}^{\mathcal{I}})$

`/* TD update */`

14 $\delta_{t+1}^{\mathcal{E}} = r_{t+1} + \gamma Q_{t+1}^{\mathcal{E}}(a_{t+1}) - Q_t^{\mathcal{E}}(a_t)$

15 $\delta_{t+1}^{\mathcal{I}} = R_t^{\phi} + \gamma Q_{t+1}^{\mathcal{I}}(a_{t+1}) - Q_t^{\mathcal{I}}(a_t)$

16 $\boldsymbol{\theta}_{t+1}^{\mathcal{E}} \leftarrow \boldsymbol{\theta}_t^{\mathcal{E}} + \alpha \delta_{t+1}^{\mathcal{E}} \mathbb{I}_{|\mathcal{A}| \times M_t}$

17 $\boldsymbol{\theta}_{t+1}^{\mathcal{I}} \leftarrow \boldsymbol{\theta}_t^{\mathcal{I}} + \alpha \delta_{t+1}^{\mathcal{I}} \mathbb{I}_{|\mathcal{A}| \times M_t}$

18 $t \leftarrow t + 1$

19 **end**

20 **return** $\boldsymbol{\theta}_{t_{end}}^{\mathcal{E}}, \boldsymbol{\theta}_{t_{end}}^{\mathcal{I}}$

M_t is the number feature observed til timestep t. We have removed the details regarding eligibility traces for brevity and clarity.

Algorithm 8 presents the final version of the algorithm that we have implemented, and for which empirical results are presented. In the algorithm shown we use the Boltzmann distributed action selection approach. We can disable Line 13 and enable the two comments below it to use the ϵ-greedy action selection approach.

In the next chapter we discuss the experimental evaluation framework we used to perform empirical evaluation. Then we showcase the state-of-the-art results that

our algorithms enjoys.

Chapter 5

Empirical Evaluation

'What can be asserted without evidence can also be dismissed without evidence.'

Christopher Hitchens

Empirical evidence is one of the fundamental requirements for validating any scientific hypothesis. In order to validate the efficacy of our exploration algorithm, this chapter showcases empirical results that represent a significant improvement over existing algorithms.

In Section 5.1 we talk about the evaluation platform and the feature set that we used to evaluate our exploration strategy. Section 5.1.1 introduces the Arcade Learning Environment (ALE) as our evaluation platform. We provides justification for choosing ALE as an environment for our agent. Further, in Section 5.1.2 we introduce the Blob-PROST feature set, and the benefits our agent enjoys from using it.

In Section 5.2 we provide the necessary foundations needed to evaluate Algorithm 8. We discuss the aspects that need to be considered in choosing a particular game for evaluation. Further, we talk about the parameters for empirical evaluation, such are number of trial, training frames, etc.

Lastly, in Section 5.3 we discuss the results of our various experiments. We compare the two action selection process discussed in Section 4.2.4 and compare their empirical performance. Then we compare the learning performance of our agent with SARSA(λ)+ϵ-greedy. Finally, we compare the evaluation scores for our agent with other leading algorithms.

5.1 Evaluation Framework

5.1.1 Arcade Learning Algorithm (ALE)

The Arcade Learning Environment (ALE) (Bellemare et al., 2013) is a software framework that interfaces with the Stella emulator (Mott and Team, 1996) for the Atari 2600 games (Montfort and Bogost, 2009). The Atari 2600 platform contains hundreds of games that vary in many aspects of game-playing such as sports, puzzle, action,

adventure, arcade, strategy etc. (Figure 5.1). Some of the games are quite challenging for human players (Bellemare et al., 2013). Due to the diverse nature of the games, a learning algorithm that can play the entire gamut of the Atari 2600 games can be considered to be generally competent. The goal of the ALE framework is to provide a platform for AI researchers to test their learning algorithm for general competence, share empirical data with the research community, and further the goal of achieving artificial general intelligence (Bellemare et al., 2013).

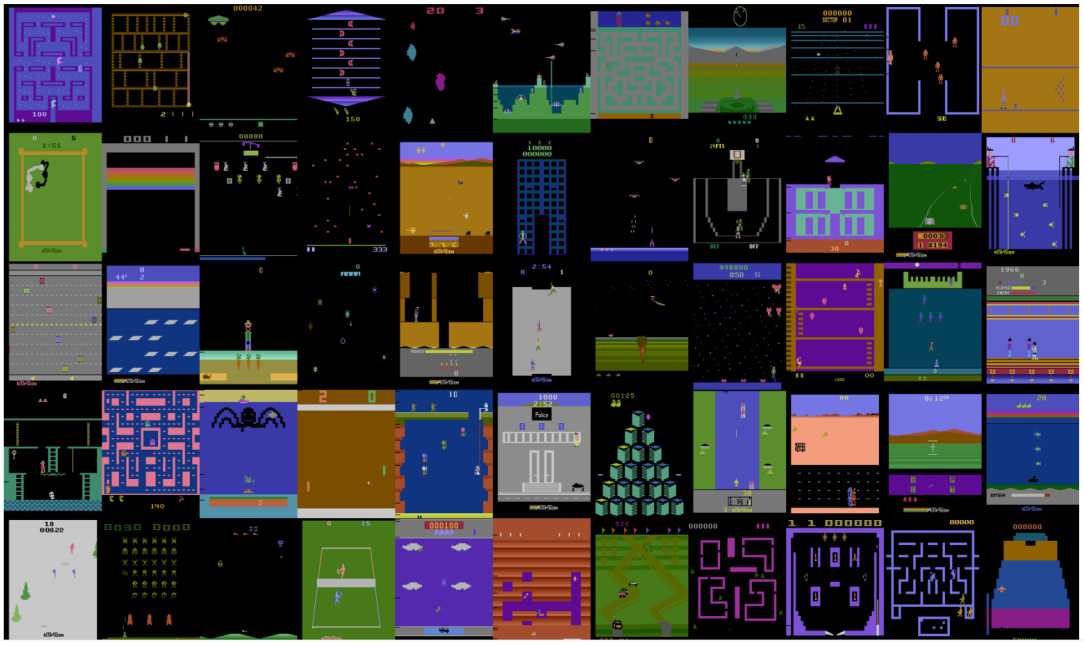

Figure 5.1: Game screens from 55 Atari 2600 games (Defazio and Graepel, 2014).

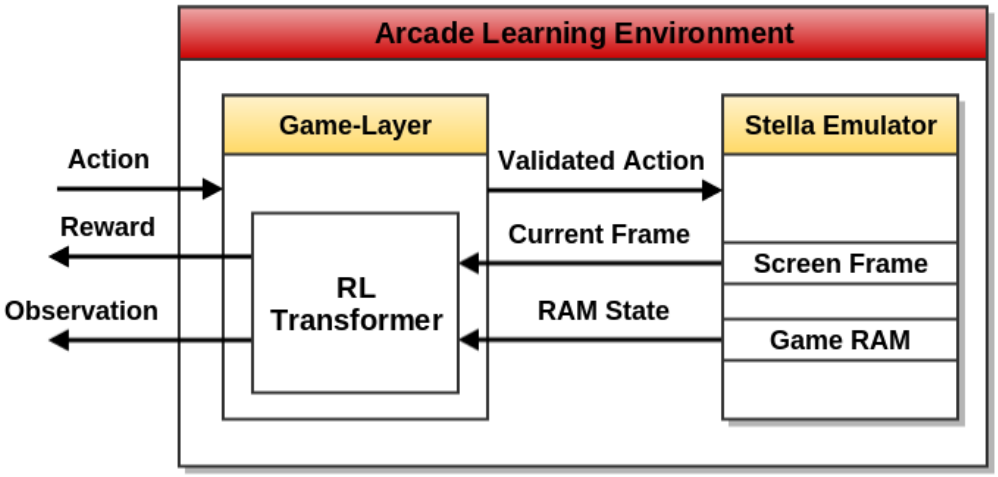

Figure 5.2: High level working of ALE for an RL algorithm.

The ALE contains a game-layer to facilitate reinforcement learning. The game-layer takes in the action from the agent and validates if it is one of the predefined 18 discrete actions. The game-layer sends the validated action to the Stella emulator which performs the action on the chosen Atari 2600 game. The resulting screen frame and the RAM state is sent to the game-layer by the emulator. Each screen frame is a 160×210 2D array, with each element representing a 7-bit pixel. The game-layer analyses the frame and RAM state to identify the game score. It then transform the game score into an appropriate reward signal expected by an RL agent. Based on the configuration settings, the game-layer returns the frame array and/or RAM state as the observation.

After the advent of the ALE, the majority of reinforcement learning publications have used ALE to provide empirical evidence (Mnih et al., 2013, 2015; Stadie et al., 2015; Osband et al., 2016; Bellemare et al., 2016; Tang et al., 2016; Ostrovski et al., 2017). Now, ALE enjoys the status as the standard test bed for testing RL algorithms. Therefore, in order to compare and contrast our results with existing research, it is crucial that we choose ALE as our evaluation platform.

5.1.2 Blob-PROST Feature Set

We observed breakthrough performance by DQN to achieve human level performance on majority of ALE games (Mnih et al., 2015). Subsequently Liang et al. (2015) did a systematic study to analyse the factors that resulted in such a dramatic increase in performance. As part of this study, they created feature sets that incorporated key representational biases encoded by DQN. One such data set is called Blob-PROST.

PROST in Blob-PROST stands for Pairwise Relative Offset in Space and Time. They argue that in most games absolute position of objects are not as important as their relative distance. Therefore, by taking into account the relative distance between two objects on screen, they are able to encode information like "there is a green pixel 5 tiles above a blue pixel". To encode the movements of objects, they take the relative distance of an object in the current frame to the object five frames in the past.

In an Atari game screen we can assume that there are many blocks of contiguous pixels with the same color. This is a common continuity assumption that is typically made in the context of computer vision. Liang et al. (2015) exploits this assumption and calls such blocks *blobs*. From a high level, Liang et al. (2015) first pre-process a frame to find blobs, and then find features based on pairwise relative distances. We refer the reader to Liang et al. (2015) for a full treatment on the construction of the feature set.

The Blob-PROST feature set contains a total of 114,702,400 binary features. Even though there are a large number of potential features, due to the sparsity of blobs, most of the features are never generated. Also, given a specific game only a relatively small number of features would be observed. Therefore, using Blob-PROST with LFA is far more computationally efficient that DQN. Moreover, empirical results from Liang et al. (2015) suggest that the fixed representations constructed in Blob-PROST has the same quality as those learned by DQN.

5.2 Empirical Evaluation

The evaluation methodology was designed to investigate and answer the following research questions:

- Does the novelty measure generalize state visit-counts when the generalization is performed in the same space (feature space) as the generalization regarding value?
- Is there performance improvement over different kinds of environments?
- How does the performance of our algorithm fare when compared to state-of-the-art Deep RL algorithms?

5.2.1 Evaluation Methodology

We evaluate Algorithm 8 in the ALE using the Blob-PROST feature set. We evaluate our algorithm on a subset of the games that are most relevant to the problem of efficient directed exploration. An important factor that determines our selection is the time and computational resources required to perform the evaluation. Finally, we perform hyperparameter sweeps to appropriately tune our algorithm.

Figure 5.3: Five games in which exploration is both difficult and crucial to performance. From top left: Montezuma's Revenge, Venture, Freeway, Frostbite, and Q*Bert.

Choosing Evaluation Games

ALE contains many games which vary in the degree to which exploration is difficult (Bellemare et al., 2016). At the lower end of the difficulty spectrum are games for which undirected exploration (ϵ-greedy) is sufficient to learn a good policy (Mnih et al., 2015). We evaluate our algorithm on hard games, i.e., games where ϵ-greedy

fails to improve substantially on a random policy. Hence we focus our evaluation on games that are classified as hard in the taxonomy provided by Bellemare et al. (2016). In their taxonomy they further split hard exploration games into games that have sparse and dense rewards.

In dense-reward games, our RL agent can expect rewards, on average, every few cycles. Dense-reward games are easier for an RL agent because policy iteration techniques such as SARSA generally require regular feedback on the agent's policy to learn; this is known as the issue of temporal credit assignment (Sutton and Barto, 1998).

On the other hand, sparse-reward games only dispense rewards infrequently. In this setting, the agent must often perform long sequences of actions in the correct order in order to receive a reward[1]. Without a good exploration strategy, stumbling upon a productive sequence of action is very challenging. Therefore, our main focus will be on sparse-reward games in which exploration is difficult.

We compare our performance with that of state-of-the-art Deep RL algorithms reported in the literature, most of which are variants of DQN (van Hasselt et al., 2016). Recall that we use LFA for value prediction, while DQN uses neural networks. As discussed previously, the Blob-PROST feature set that we selected for LFA has the property that it closely models the representations learned by DQN (Liang et al., 2015). Since the Blob-PROST feature set has this property, it is appropriate to compare our ϕ-EB algorithm to the other algorithms that use DQN. Now from the empirical data presented in Liang et al. (2015), we look for games that perform similar to DQN, and meet the exploration difficult criterion outlined above.

With these in mind, we choose the following five hard exploration games to evaluate our exploration strategy.

- Sparse Reward Games (Figure 5.3, Top Layer)
  - Montezuma's Revenge
  - Venture
  - Freeway
- Dense Reward Games (Figure 5.3, Bottom Layer)
  - Frostbite
  - Q*bert

Computational Roadblocks

The computational requirement for games in the ALE are very demanding, especially in the absence of graphical processing units (GPUs), which excel at the dense linear algebra computations common in vision-related tasks (Liang et al., 2015). Given the high-dimensional nature of the problem, agents must be trained for several days on

[1]Classic examples of such games are the board games Chess and Go; rewards are only dispensed once, at the very end of the game.

end to obtain satisfactory performance. Frostbite and Q*bert were especially computationally intensive, and we had to train for several weeks to obtain sufficient data. Given our limited time and computational resources, we had to place the following constraints on our evaluation.

- In Section 4.2.1 we remarked that due to sparse nature of the feature vector our algorithm runs in $O(M_t)$ where M_t is the number of unique features observed till time t. This means that different games run at different speeds. We trained our algorithm on all games for 100 million frames except for Q*bert which was trained only for 80 million frames.

- Our main focus is to showcase performance gains in sparse reward hard exploration games. Therefore we focused bulk of our computational resources into running multiple trial for Montezuma's Revenge and Venture. This is where our algorithm leads other state-of-the-art exploration strategies.

Tuning the hyper-parameter β

We performed independent evaluation of all the chosen games for discrete values of β in the range $[0.0001, 0.5]$. Recall that β is a parameter that controls the magnitude of the exploration bonus. We observed that $\beta = 0.05$ is the best performing value for all the games except for Freeway. Recall that because of the nature of the game, there is a large number of unique Blob-PROST features active. If β is high enough, the chicken just remains stationary and receives novelty rewards for observing all the changes in the traffic. When we set $\beta = 0.035$, our agent performed much better and delivered comparable results with the baseline algorithm.

Training Methodology

Training and evaluation of learning algorithms in high-dimensional spaces is computationally demanding. Due to the constantly evolving nature of the field, there is no general consensus on how many cycles is required to train an agent. Some of the major exploration algorithms published recently report training till 200 million frames. Due to limited time and computational resources, this amount of training is not feasible for us.

We perform empirical evaluation of Montezuma's Revenge and Venture for five independent trials, and two independent trials for Frostbite, Freeway, and Q*bert. With the exception of Q*bert, all agents are trained for 100 Million frames, and then evaluated for 500 episodes. The result for Q*bert is reported after training for 80 million frames with subsequent 500 episodes of training.

We use the average score per episode, which is a common metric used to report scores (Bellemare et al., 2013; Mnih et al., 2015; Bellemare et al., 2016).

5.2.2 Sparse Reward Games

Montezuma's Revenge

Montezuma's Revenge is widely regarded as one of the most difficult games in the Atari 2600 suite. Learning algorithms typically suffer here due to the problem of long term credit assignment and sparse rewards. For example, DQN with ϵ-greedy exploration achieves a score of 0 after training for 200 million frames (Mnih et al., 2015). In order to get the very first reward of the game the agent must climb down a ladder, jump onto the pole, jump onto a raised platform, climb down a ladder, walk left and jump to avoid an enemy, climb another ladder, and finally obtain a golden key. This long sequence of complex actions is required to simply achieve the first reward in the first of 24 rooms, arranged in a pyramid structure (Figure 5.4). It is evident from the complexity of the game that a random exploration strategy will fail miserably. The challenges posed by this game make the game central to our evaluation.

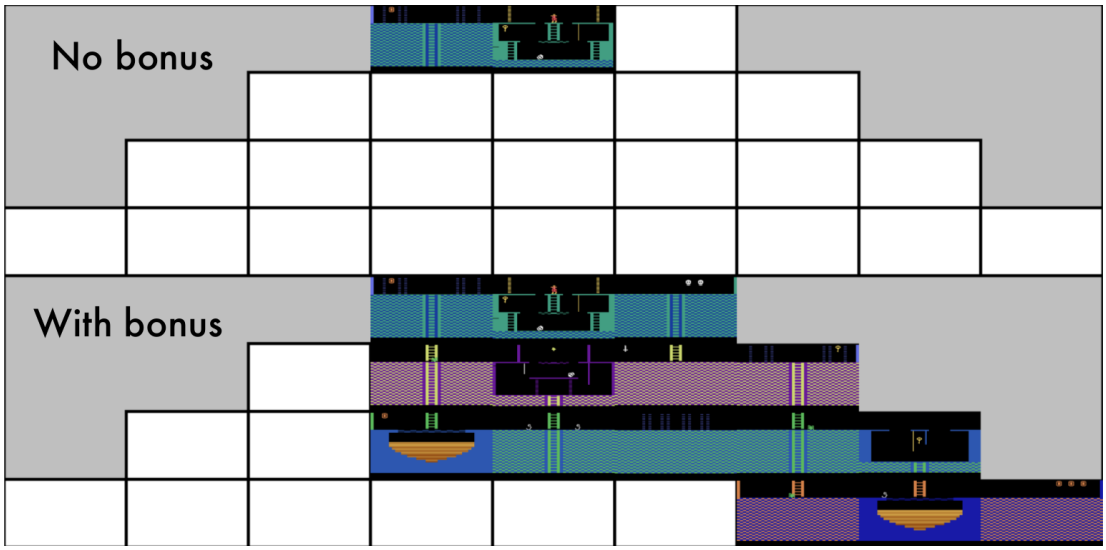

Figure 5.4: Montezuma's Revenge: Rooms visited by undirected exploration (DQN+ϵ-greedy, above) vs. directed exploration (DQN+CTS-EB, below) (Bellemare et al., 2016).

Our agent showed steady learning progress and was relatively quick to learn the initial sequence of actions needed to get the golden key. Analysis from the logs showed increasing novelty measure associated with novel events. For example, one of the rooms had a dead-end for one side, but there was a reward there. When our agent reached the dead-end and received the reward once, it kept getting stuck in that region for a few episodes. During the episodes where the agent was stuck, we could observe the novelty values of the other end slowly rising. After a few episodes the agent started going in the opposite direction based on the novelty rewards it was

getting, and proceeded to explore further.

In Montezuma's revenge, the total number of rooms visited is a good measure of exploration efficiency (Bellemare et al., 2016). Prior to the work of Bellemare et al. (2016) no agent had visited more that 3 rooms without domain specific tailoring. Bellemare et al. (2016)'s agent visited 15 room while our agent visited 14 rooms in total (Figure 5.4). Both Bellemare et al. (2016)'s and our agent enjoy a peak score of **6600**, which is the highest reported score.

Venture

Venture is another hard, sparse reward exploration game with complex visual representations. Figure 5.7 represents the two different visual states of venture. In Figure 5.5 the agent is depicted by the small tiny pink pixel towards the bottom middle of the screen. When the agent is in the outer level it is powerless. The only way to stay alive is to navigate the maze and/or entering one of the rooms to avoid the green goblins. Once the agent enters the room, the screen changes to Figure 5.6. Now our agent is transformed into a smiley face with the ability to fire a projectile weapon. Here the agent is chased by evil blue robots which can be destroyed by the agent's weapon. The small tiny cup-like structure in the bottom left corner of the room is the goal and the first reward. Destroying the blue monsters does not result in any reward. The game presents some intricate dynamics, with a large state-space to explore, and so is difficult for a naive learning algorithm. Therefore Venture is a must-have game in our evaluation roster.

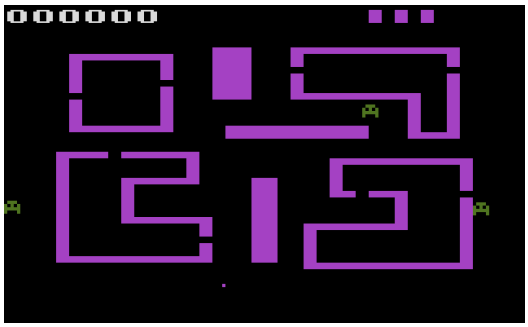

Figure 5.5: Venture Outer Level

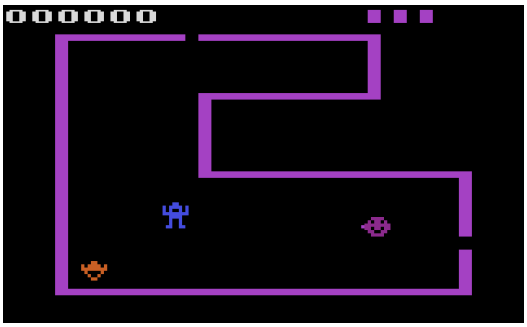

Figure 5.6: Venture Inner Level

Figure 5.7: Two visual states of venture (Atari 2600)

The agent initially moves around the black region to the bottom of the screen where it starts out (Figure 5.5). The novelty of the agent with empty black space around it quickly reduces. We observed that the agent starts to hug the outer walls of the rooms. The room walls together with the agents are novel features. Then the agent moves along the room wall towards the entry of the room. When the agent enters the room the screen now transforms as shown in Figure 5.6. We later observe

that, with subsequent visits to the lower room our agent learned to shoot and kill the blue robots and get the reward. Our agent achieves substantial improvement over Bellemare et al. (2016). Even though they do not report their evaluation score for Venture, Ostrovski et al. (2017) reports the score of DQN+CTS-EB as 82.2, whereas our evaluation score is 1169.2.

Freeway

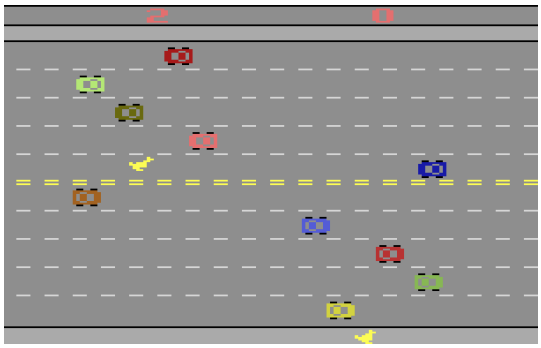

Figure 5.8: Freeway (Atari2600)

Here the agent is a chicken that must cross a busy highway (Figure 5.3 Top layer, third game). The agent receives the reward only when it reaches the other end of the road. Being hit by a car sets you back some positions down the road. Even though this is not a complex game, it is still a hard exploration game. In freeway, cars are always moving from left to right in every frame. Therefore, there is a large number of active unique Blob-PROST features.

When tuning the hyper-parameter β we talked about the large number of unique Blob-PROST features created due to the constant movement of traffic. These changing unique features constantly floods the agent with high novelty rewards. Therefore the agent is willing to just stand and observe the traffic. When we reduced the exploration coefficient β, the agent's extrinsic reward is no longer overwhelmed by the novelty reward. Also, due to the initial optimism, the agent is encouraged to move upwards. Once the agent manages to cross the road once, it reinforces the up action and there the agent starts learning faster.

5.2.3 Dense Reward Games

Q*bert

Figure 5.9: Qbert (Atari2600)

In Q*Bert, the agent stands on a pyramid of cubes (Figure 5.9). The goal of the agent is to jump on all the cubes without falling off the edge or being captured by an adversary. When the agent has highlighted all the cubes by jumping on them at least once, the level is cleared. The game has multiple levels. On higher levels, the task still remain the same, but the enemies become smarter, making it increasingly difficult to accomplish the task while avoiding capture. As discussed in Section 3.1.1 the only other difference between levels is the choice of color scheme.

Frostbite

Figure 5.10: Frostbite (Atari2600)

In Frostbite, the agent is an Eskimo jumping up and down on an ice platform which magically results in the building of an igloo (Figure 5.10). Each time the agent jumps on a pure white ice platform, the agent receives a reward. The agents has to consistently do this by avoiding obstacles/dangers, and picking up bonus points. Once the igloo has been built the agent need to perform a level-end move by entering into the igloo. Since there are multiple ways of maximizing the score, and the existence of a

level-end move makes it a hard exploration game. Our main aim with this game is to confirm that our exploration strategy improves upon the baseline performance.

5.3 Results

From here on we denote our agent SARSA(λ)+$\boldsymbol{\phi}$-EB as SARSA-ϕ-EB and SARSA(λ)+ϵ-greedy as SARSA-ϵ for notational brevity.

5.3.1 Boltzmann vs. ϵ-greedy Action Selection

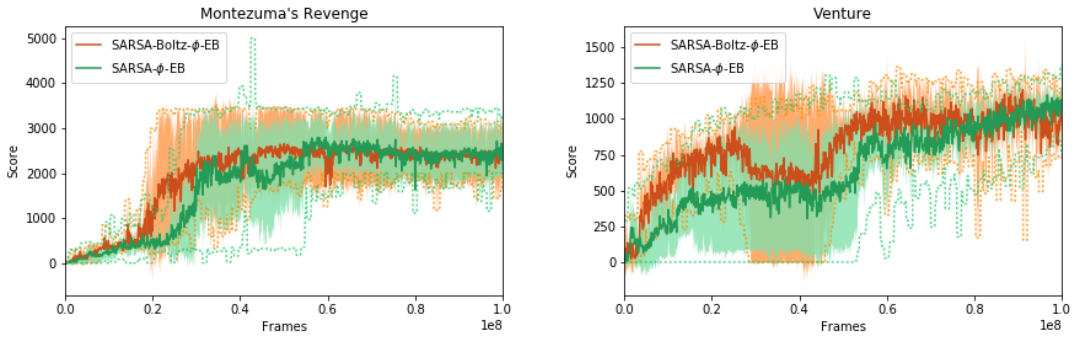

Figure 5.11: Average training score for Action Selection using Boltzmann vs. ϵ-greedy. Shaded regions describe one standard deviation. Dashed lines represent min/max scores.

Recall that in Section 4.2.4 we discussed the problem of action selection and presented two methods that introduce stochasticity into the agent's policy. Here we compare two agents differing only in the action selection functionality. SARSA-Boltz-ϕ-EB introduces stochasticity via a Boltzmann-distributed random action, whereas SARSA-ϕ-EB does via ϵ-greedy.

Figure 5.11 shows the learning curve for the two agent evaluated for the games Montezuma's Revenge and Venture. From the plots we can observe that during the initial training period Boltzmann-distributed action selection has an advantage over epsilon-greedy. In the long run both have essentially the same average score. This behavior is expected since ϵ is annealed with training cycles, and hence with time the agent takes fewer purely exploratory actions.

What we hoped to see in this experiment is for the Boltzmann distributed action selection to send the agent into a steeper learning trajectory. Unfortunately, this experiment concludes that there is no long term gain to having a Boltzmann-distributed action selection process.

5.3.2 Comparison with ϵ-greedy

Overview

Figure 5.12 shows the comparison between learning curves for our agent SARSA-ϕ-EB, and the benchmark implementation SARSA-ϵ. The plots clearly illustrates that SARSA-ϕ-EB significantly outperforms SARSA-ϵ on both Montezuma's Revenge and Venture; two of hardest exploration games in the ALE. In Q*bert and Frostbite SARSA-ϕ-EB consistently outperforms SARSA-ϵ, but not by a huge margin. Frequent rewards from these games give a constant feedback to SARSA-ϵ, helping it to chart a positive learning path. With $\beta = 0.05$ freeway fails to obtain any score, but with $\beta = 0.035$ SARSA-ϕ-EB achieved a marginally better performance that the baseline SARSA-ϵ.

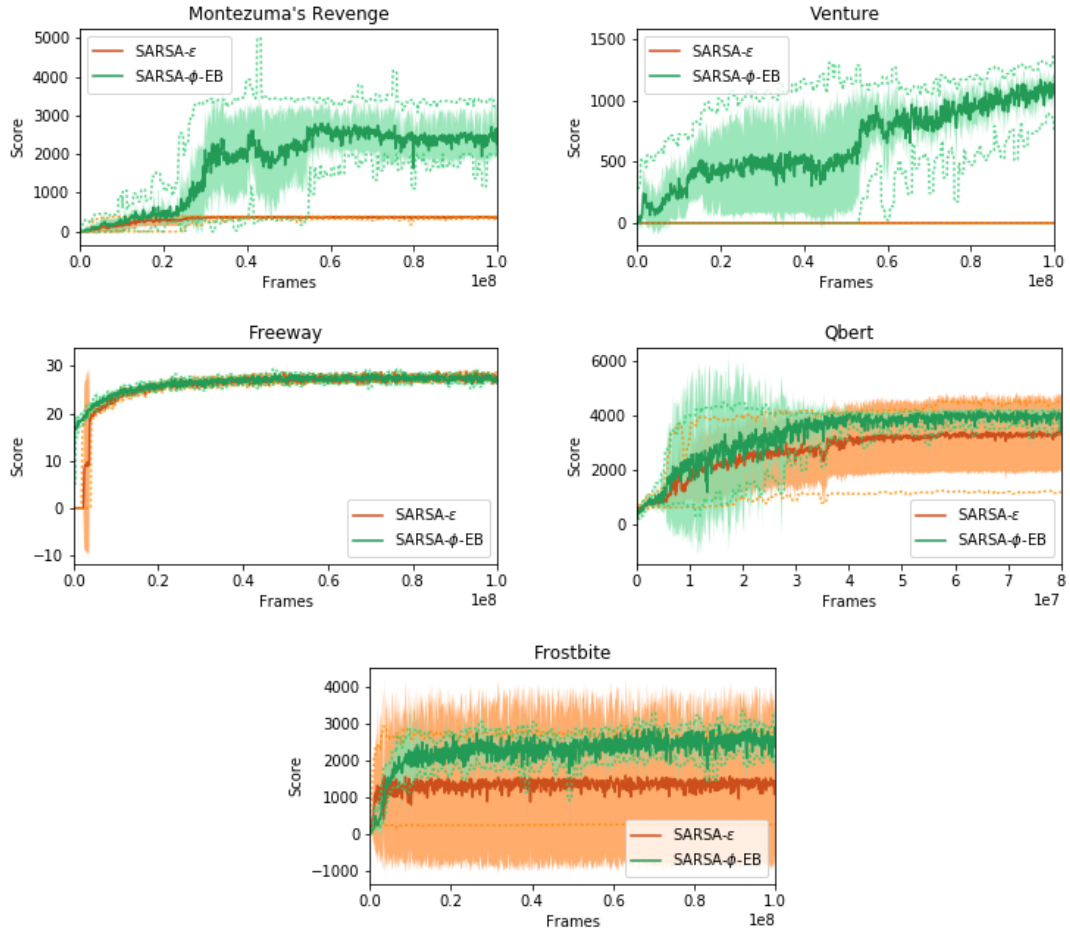

Figure 5.12: Average training score for SARSA-ϕ-EB vs. SARSA-ϵ. Shaded regions describe one standard deviation. Dashed lines represent min/max scores. (Martin et al., 2017)

Montezuma's Revenge

After an average of 20 Million frames the policy of SARSA-ϵ converges. It learns the policy to consistently get the golden key, buy rarely leaves the room. Even when SARSA-ϵ leaves the room it is always to the left room and gets immediately killed by the laser beams. SARSA-ϵ never learns how to get through the laser beams. SARSA-ϕ-EB performs remarkably well, visiting 14 rooms in total and observing a peak score of 6600 around the 40 Million frame mark. SARSA-ϕ-EB learns to go through laser doors by timing the movements perfectly, avoid dead-ends, duck and jump over skulls, etc.

Venture

SARSA-ϵ fails to score any points while SARSA-ϕ-EB performs exceptionally well in Venture. It is the most impressive performance gain in this evaluation. SARSA-ϕ-EB can quickly identify that spending time in the black region is not novel. It is attracted (because of novelty) to the walls of the room quickly. The agent then moves along the wall until it sees a nearby opposite wall or the room entrance. Once the agent starts entering a room consistently, it quickly learns how to attack blue robots and obtains the reward. The positive learning trend suggest that even higher scores are possible with more training.

Freeway

With $\beta = 0.05$ our agent SARSA-ϕ-EB fails to cross the road for reasons previously mentioned in Section 5.2 and Section 5.2.2. But with $\beta = 0.035$, SARSA-ϕ-EB explores quickly initially and received the reward by crossing the road for the first time. Due to the undirected nature of exploration, SARSA-ϵ has to first random walk across the road to receive a reward. Once both the algorithms receive a reward they perform similarly, with SARSA-ϕ-EB doing marginally better.

Recall that the components of SARSA-ϕ-EB were designed such that there is only a single point of change from SARSA-ϵ. The only difference between SARSA-ϕ-EB and SARSA-ϵ is our exploration module, $\boldsymbol{\phi}$-EB. Hence we can conclude that the empirical performance gains is solely due to the introduction of our exploration algorithm. Further we can conclude that $\boldsymbol{\phi} - EB$ is significantly better that ϵ-greedy in sparse reward hard exploration domains, and consistently outperforms ϵ-greedy in all domains.

5.3.3 Comparison with Leading Algorithms

Table 5.1 shows the evaluation score for the final policy learnt by our agent. In order to compare and contrast the efficacy of our algorithm, Table 5.1 also presents the evaluation scores reported by leading RL algorithms.

	Venture	Montezuma	Freeway	Frostbite	Q*bert
Sarsa-ϕ-EB	1169.2	2745.4	0.0	2770.1	4111.8
Sarsa-ϵ	0.0	399.5	29.9	1394.3	3895.3
DQN+CTS-EB	N/A	3459	N/A	N/A	N/A
A3C+	0	142	27	507	15805
MP-EB	N/A	0	12	380	N/A
DDQN	98	0	33	1683	15088
Dueling	497	0	0	4672	19220
DQN-PA	1172	0	33	3469	5237
Gorila	1245	4	12	605	10816
TRPO	121	0	16	2869	7733
TRPO-Hash	445	75	34	5214	N/A

Table 5.1: Average evaluation score for leading algorithms. Sarsa-ϕ-EB and Sarsa-ϵ were evaluated after 100M training frames on all games except Q*bert, for which they trained for 80M frames. All other algorithms were evaluated after 200M frames. (Martin et al., 2017)

Our agent reports an average evaluation score of 2745.4 on Montezuma's Revenge. This score is the second-highest reported score for Montezuma, the leading score reported for DQN+CTS-EB by Bellemare et al. (2016). Algorithms such as MP-EB (Stadie et al., 2015), TRPO-Hash (Tang et al., 2016), A3C+ (Bellemare et al., 2016) does not score more than 200 points in Montezuma's Revenge, despite the presence of an exploration strategy module.

SARSA-ϕ-EB evaluated on Venture reported a score of 1169.2, the third-highest reported score. Again it outperforms all the other exploration algorithms. Bellemare et al. (2016) does not report their score for Venture. A recent evaluation of DQN+CTS-EB done by Ostrovski et al. (2017) reported that DQN+CTS-EB obtained a score of 82.2 on venture.

It is known that non-linear algorithms perform much better on Q*bert therefore our results is not surprising (Wang et al., 2015). Frostbite is notoriously slow to train because of the large number of features. Given the dense reward nature of the game and the small training time, we achieve competitive results.

Our results for Montezuma and Venture are extremely competitive and can be considered state-of-the-art. The key observation from Table 5.1 is this, if we look at the score of Montezuma and Venture together, ours is the only algorithm that achieve state-of-the-art results in both the games.

Chapter 6

Conclusion and Future Work

In this thesis we have presented the ϕ-Exploration Bonus (ϕ-EB) method, a novel approach to perform directed exploration in large problem domains. The algorithm is simple to implement, and is compatible with any value-based RL algorithm that uses Linear Function Approximation (LFA) to predict value. Our method also enjoys lower computational requirements when compared to other leading exploration strategies. Our empirical evaluation demonstrates that measuring novelty in feature space is a simple and effective way to drive efficient exploration on MDPs of practical interest. It also lends support to our hypothesis that defining a novelty measure in feature space is a principled way to generalize state-visit counts to large problems. In contrast to other approaches, measuring novelty in feature space avoids building an exploration-specific state-representation. Instead, our method exploits the task-relevant features that are already being used for value estimation.

There are myriad ways in which this work could be extended, and the problem of efficient exploration in large MDPs is still wide open. A promising direction for future research would be a rigorous empirical comparison of the various generalized count-based algorithms which we have discussed. At present, many of the reported results are not helpful in deciding which is the better approach to measuring novelty or generalizing visit-counts. Different exploration algorithms are presented in conjunction with totally different value-estimation methods, and it can be difficult to discern whether or not the exploration method used is responsible for the quality of the results. More theoretical understanding of the problem of exploration in the high-dimensional setting is also sorely needed, and we hope to build upon the results presented here in future work.

Bibliography

Barto, A. and Dietterich, T., 2004. Reinforcement learning and its relationship to supervised learning. *Handbook of Learning and Approximate Dynamic Programming*, (2004), 47–64. (cited on page 1)

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M., 2013. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47 (7 2013), 253–279. doi:10.1613/jair.3912. (cited on pages 24, 34, 45, 46, and 50)

Bellemare, M. G.; Srinivasan, S.; Ostrovski, G.; Schaul, T.; Saxton, D.; and Munos, R., 2016. Unifying Count-Based Exploration and Intrinsic Motivation. *CoRR*, abs/1606.0 (6 2016), 1–26. (cited on pages xiii, 3, 12, 14, 15, 17, 20, 21, 25, 26, 33, 34, 41, 47, 48, 49, 50, 51, 52, 53, and 58)

Bertsekas, D. P. and Tsitsiklis, J. N., 1996. *Neuro-Dynamic Programming*, vol. 5. Athena Scientific. ISBN 1886529108. doi:10.1109/MCSE.1998.683749. (cited on pages 7 and 9)

Bishop, C. M., 2007. Pattern Recognition and Machine Learning. *Journal of Electronic Imaging*, 16, 4 (1 2007), 049901. doi:10.1117/1.2819119. (cited on page 1)

Dearden, R.; Friedman, N.; and Russell, S., 1998. Bayesian Q-learning. *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, (1998), 761–768. (cited on page 15)

Defazio, A. and Graepel, T., 2014. A Comparison of learning algorithms on the Arcade Learning Environment. *CoRR*, abs/1410.8 (10 2014). (cited on page 46)

Duff, M., 2002. *Optimal Learning: Computational procedures for Bayes-adaptive Markov decision processes*. Ph.D. thesis, University of Massachusetts at Amherst. (cited on page 16)

Frank, M.; Leitner, J.; Stollenga, M.; Förster, A.; and Schmidhuber, J., 2014. Curiosity driven reinforcement learning for motion planning on humanoids. *Frontiers in Neurorobotics*, 7, JAN (2014), 1–15. doi:10.3389/fnbot.2013.00025. (cited on page 16)

Hutter, M., 2005. *Universal Artificial Intellegence*. Texts in Theoretical Computer Science An EATCS Series. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-540-22139-5. doi:10.1007/b138233. (cited on page 5)

Hutter, M., 2013. Sparse Adaptive Dirichlet-Multinomial-like Processes. *Journal of Machine Learning Research*, 30 (5 2013). (cited on page 25)

Kakade, S. M., 2003. On the Sample Complexity of Reinforcement Learning. In *International Conference on Machine Learning*, 133. doi:10.1.1.164.7844. (cited on page 14)

Kolter, J. Z. and Ng, A. Y., 2009. Near-Bayesian exploration in polynomial time. In *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*, 1–8. ACM Press, New York, New York, USA. doi:10.1145/1553374.1553441. (cited on page 14)

Krichevsky, R. E. and Trofimov, V. K., 1981. The Performance of Universal Encoding. *IEEE Transactions on Information Theory*, 27, 2 (1981), 199–207. doi:10.1109/TIT.1981.1056331. (cited on page 25)

Lai, T. and Robbins, H., 1985. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6, 1 (3 1985), 4–22. doi:10.1016/0196-8858(85)90002-8. (cited on page 14)

Lake, B. M.; Ullman, T. D.; Tenenbaum, J. B.; and Gershman, S. J., 2016. Building Machines That Learn and Think Like People. *Behavioral and Brain Sciences*, (11 2016), 1–101. doi:10.1017/S0140525X16001837. (cited on page 16)

Leike, J., 2016. Exploration Potential. *CoRR*, abs/1609.0 (9 2016), 1–9. (cited on page 16)

Liang, Y.; Machado, M. C.; Talvitie, E.; and Bowling, M., 2015. State of the Art Control of Atari Games Using Shallow Reinforcement Learning. *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, (12 2015), 485–493. (cited on pages 24, 32, 33, 47, and 49)

Martin, J.; Narayanan S, S.; Everitt, T.; and Hutter, M., 2017. Count-Based Exploration in Feature Space for Reinforcement Learning. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*. AAAI Press. (cited on pages iii, xiii, xv, 56, and 58)

Melo, F. S.; Meyn, S. P.; and Ribeiro, M. I., 2008. An analysis of reinforcement learning with function approximation. *Proceedings of the 25th international conference on Machine learning*, (2008), 664–671. doi:10.1145/1390156.1390240. (cited on page 33)

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M., 2013. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602*, (12 2013). (cited on pages 34 and 47)

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg,

S.; AND HASSABIS, D., 2015. Human-level control through deep reinforcement learning. *Nature*, 518, 7540 (2015), 529–533. doi:10.1038/nature14236. (cited on pages 34, 47, 48, 50, and 51)

MOHAMED, S. AND REZENDE, D. J., 2015. Variational Information Maximisation for Intrinsically Motivated Reinforcement Learning. *ArXiv e-prints*, (9 2015). (cited on page 16)

MONTFORT, N. AND BOGOST, I., 2009. Racing the beam: the Atari Video computer system. *Platform studies*, (2009), xii, 180 p. doi:10.1162/leon.2010.43.2.188. (cited on page 45)

MOODY, J. AND SAFFELL, M., 2001. Learning to trade via direct reinforcement. *IEEE Transactions on Neural Networks*, 12, 4 (2001), 875–889. doi:10.1109/72.935097. (cited on page 23)

MOTT, B. W. AND TEAM, S., 1996. Stella: a multiplatform Atari 2600 VCS emulator. (cited on page 45)

OSBAND, I.; BLUNDELL, C.; PRITZEL, A.; AND VAN ROY, B., 2016. Deep Exploration via Bootstrapped DQN. *Advances In Neural Information Processing Systems*, (2 2016). (cited on pages 14, 34, and 47)

OSBAND, I. AND VAN ROY, B., 2016. Why is Posterior Sampling Better than Optimism for Reinforcement Learning. *arXiv preprint*, (7 2016). (cited on page 12)

OSTROVSKI, G.; BELLEMARE, M. G.; OORD, A. V. D.; AND MUNOS, R., 2017. Count-Based Exploration with Neural Density Models. *arXiv preprint*, (3 2017). (cited on pages 15, 34, 47, 53, and 58)

OUDEYER, P.-Y., 2007. What is intrinsic motivation? A typology of computational approaches. *Frontiers in Neurorobotics*, 1, NOV (2007), 1–14. doi:10.3389/neuro.12.006.2007. (cited on page 16)

PUTERMAN, M., 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. John Wiley & Sons, Inc., Hoboken, NJ, USA, 1 edn. ISBN 9780470316887. doi:10.1002/9780470316887. (cited on page 5)

SCHMIDHUBER, J., 1991. A Possibility for Implementing Curiosity and Boredom in Model-Building Neural Controllers. In *From Animals to Animats:Proceedings of the First International Conference on Simulation of Adaptive Behavior*, 222–227. MIT Press, Paris, France. ISBN 0-262-63138-5. (cited on page 16)

SCHMIDHUBER, J., 2010. Formal Theory of Creativity, Fun, and Intrinsic Motivation. *IEEE Transactions on Autonomous Mental Development*, 2, 3 (9 2010), 230–247. doi:10.1109/TAMD.2010.2056368. (cited on page 16)

Singh, S.; Jaakkola, T.; Littman, M. L.; Szepes, C.; and Hu, A. S., 2000. Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms. *Machine Learning*, 39, 1998 (2000), 287–308. (cited on page 11)

Spelke, E. S. and Kinzler, K. D., 2007. Core knowledge. *Developmental Science*, 10, 1 (1 2007), 89–96. doi:10.1111/j.1467-7687.2007.00569.x. (cited on page 16)

Stadie, B. C.; Levine, S.; and Abbeel, P., 2015. Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models. *arXiv*, (2015), 1–11. (cited on pages 20, 34, 47, and 58)

Steunebrink, B. R.; Koutník, J.; Thórisson, K. R.; Nivel, E.; and Schmidhuber, J., 2013. Resource-Bounded Machines are Motivated to be Effective, Efficient, and Curious. In *Artificial General Intelligence: 6th International Conference, AGI 2013, Beijing, China, July 31 – August 3, 2013 Proceedings*, vol. 7999 LNAI, 119–129. Springer Berlin Heidelberg. ISBN 978-3-642-39521-5. doi:10.1007/978-3-642-39521-5{_}13. (cited on page 16)

Strehl, A. L. and Littman, M. L., 2004. An empirical evaluation of interval estimation for Markov decision processes. In *Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI*, 128–135. doi:10.1109/ICTAI.2004.28. (cited on page 14)

Strehl, A. L. and Littman, M. L., 2008. An analysis of model-based Interval Estimation for Markov Decision Processes. *Journal of Computer and System Sciences*, 74, 8 (12 2008), 1309–1331. doi:10.1016/j.jcss.2007.08.009. (cited on page 14)

Sutton, R. and Barto, A., 1998. *Reinforcement Learning: An Introduction*, vol. 1. MIT press Cambridge. ISBN 0262193981. (cited on pages xiii, 1, 2, 7, 8, 9, 10, 11, 23, 32, and 49)

Sutton, R. S., 1988. Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3, 1 (1988), 9–44. doi:10.1023/A:1022633531479. (cited on page 6)

Sutton, R. S., 1990. Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. *ML*, (1990), 216–224. doi:10.1.1.51.7362. (cited on page 14)

Sutton, R. S., 1999. Reinforcement Learning: Past, Present and Future. In *Asia-Pacific Conference on Simulated Evolution and Learning*, 195–197. Springer. doi:10.1007/3-540-48873-1{_}26. (cited on page 1)

Tang, H.; Houthooft, R.; Foote, D.; Stooke, A.; Chen, X.; Duan, Y.; Schulman, J.; De Turck, F.; and Abbeel, P., 2016. #Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning. *arXiv preprint arXiv:1611.04717*, (2016), 1–11. (cited on pages 15, 21, 34, 47, and 58)

Thrun, S. B., 1992. Efficient Exploration In Reinforcement Learning. In *Science*, January, 1–44. (cited on pages 12 and 16)

van Hasselt, H.; Guez, A.; and Silver, D., 2016. Deep Reinforcement Learning with Double Q-learning. In *AAAI*, 2094–2100. (cited on page 49)

Veness, J.; Ng, K. S.; Hutter, M.; and Bowling, M., 2012. Context tree switching. In *Data Compression Conference Proceedings*, 327–336. doi:10.1109/DCC.2012.39. (cited on page 15)

Wang, Z.; Schaul, T.; Hessel, M.; van Hasselt, H.; Lanctot, M.; and de Freitas, N., 2015. Dueling Network Architectures for Deep Reinforcement Learning. *CoRR*, abs/1511.0, 9 (11 2015), 1–16. doi:10.1109/MCOM.2016.7378425. (cited on page 58)

Watkins, C. J. C. H. and Dayan, P., 1992. Q-learning. *Machine Learning*, 8, 3-4 (1992), 279–292. doi:10.1007/BF00992698. (cited on page 9)