# An Empirical Study of Exploration Strategies for Model-Free Reinforcement Learning

by

Nikolaus Winget Yasui

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

# Abstract

Reinforcement Learning is a formalism for learning by trial and error. Unfortunately, trial and error can take a long time to find a solution if the agent does not efficiently explore the behaviours available to it. Moreover, how an agent ought to explore depends on the task that the agent is trying to learn. In this thesis we study how an agent's exploration strategy affects how quickly it learns to solve different problems. In particular, we focus on model-free algorithms that learn value functions. We first examine the space of problems, or environments, that reinforcement learning agents are expected to solve. We identify six properties of environments that can make exploration difficult, and design a prototypical environment expressing each property. We also survey the exploration literature and categorize existing exploration methods by the heuristic that guides their behaviour. Lastly, we conduct an empirical study evaluating the performance of several exploration methods on our prototypical exploration environments. We found that only one method, Linear Upper Confidence Least Squares, was able to consistently perform well in every environment. We also found that methods which add a bonus to their value function tended to explore much more effectively than methods which add a bonus to their rewards. Our investigation of value-based exploration provides a novel, systematic approach to understanding the strengths and weaknesses of exploration algorithms in reinforcement learning.

# Acknowledgements

I would first like to thank my supervisor Martha White for her invaluable support and advice. She encouraged my diverse research interests and was always able to help when I needed it. The research atmosphere she created was truly fantastic.

I would also like to acknowledge the other professors in the department for supporting student research. In particular, Adam White provided substantial encouragement, support, and feedback during my degree. James Wright also introduced me to several interesting topics, and has been incredibly patient while waiting for me to start my PhD. I would also like to acknowledge NSERC for providing funding during my degree, and Matthew Pietrosanu from the University of Alberta Training and Consulting Center for providing statistical advice.

My thanks goes to the members of the RLAI lab, with whom I have spent countless hours discussing various problems in research and in life. Without those lively discussions, my time at the University of Alberta would be much less interesting and fulfilling. Thank you for everything.

Finally, I would like to thank my family, my partner Ashley, and her family for their continued love and support throughout my degree. The majority of this thesis was written under their care and hospitality, and I am very fortunate to have them in my life.

I am very thankful for all the support I have received and I look forward to passing it on to others in the future.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Technology improves the efficiency of production. Recently, we have begun to study technologies that not only perform useful tasks in our lives, but automatically learn how to perform those tasks on their own. At the center of these advances is reinforcement learning (RL), a computational framework for learning that captures a huge variety of tasks and suggests how algorithms might learn to perform those tasks. Thanks to RL, algorithms have learned how to play board games like chess, go, and shogi at a high level (Silver et al. 2017). RL is popular enough and effective enough that large technology companies such as Amazon and Facebook have released RL development platforms and products (Facebook 2020; Services 2020).

Reinforcement learning describes tasks as an interaction between a learning agent and an environment. The agent is an algorithm, either running on a computer, or a brain, or a robot. The environment can be anything that the agent interacts with, from a simulation, to a game, to the real world. Consider an agent that learns to drive a car. The agent is an algorithm running on a computer within the car. The environment is the real world, where the car is driving. At each point in time the agent chooses an action, such as how to accelerate and steer, in response to the current situation.

Importantly, the agent is not set free in the environment to behave without purpose. Its goal is to maximize a reward signal produced by the environment, which typically depends on the situation and the agent's action. The reward signal describes the most important properties of the behaviour

the human designer wishes the agent to learn to the agent. In games, the reward is often the agent's score, or a simply binary indicator of whether the agent won which is only provided the end of the game. In a self-driving car the reward would be larger if the agent drives well, and smaller when the agent drives poorly or dangerously. To allow agents to find solutions that are better than those already used by humans, the reward signal is usually only non-zero in situations that are unambiguously good or bad. Moreover, the agent does not know what the reward will be in every situation. So the agent must explore the environment to identify which actions and which situations will produce large reward. In general, changing the reward signal changes the task that the agent learns to perform, even if the rest of the environment is kept fixed. Together, the set of situations, actions, and rewards completely describes an environment and its associated learning problem.

It is natural to imagine that an agent could learn how to do something in the same way that humans do, namely by attempting to do it and learning from mistakes that are made along the way. In a general sense, this is how all the algorithms we study behave. As a field, we typically think of the problem of learning to perform a task as two subproblems: that of learning, and that of performing the task. For example, consider how a person behaves as they learn how to cook. Sometimes, they experiment with flavors and techniques to learn how they impact the final dish. Other times, they do their best to cook something delicious. If the person always tries to cook delicious food, they will lose out on critical learning experiences that might substantially improve their cooking in the long run. A person might be able to intuitively identify the cooking experiments that will help them learn quickly, but developing an algorithm that learns as quickly is another challenge altogether.

Part of the challenge in developing learning algorithms is that they should be applicable to a huge variety of problem settings — for example, an algorithm that can only learn to predict stock prices is of limited use to society. To ensure that algorithms can learn to do many tasks, RL uses an expressive mathematical language to describe environments. Unfortunately, possessing

the language and understanding the environments that it can express are quite different. We are especially interested in understanding why some environments are more difficult for agents to explore than others. While some authors have tried to identify environments that embody disparate challenges for RL agents (Langford 2018; Osband, Doron, et al. 2019), understanding the relationship between an agent and its environment remains an exciting area of research. We present several properties of environments that make them easier or harder to explore in Chapter 3. These properties will help researchers develop algorithms that learn efficiently in a wider variety of problem settings.

Just as it is useful to understand what kinds of environments exist and how they impact learning, it can also help researchers to understand the different heuristics that can guide exploration. There is a huge number of exploration methods, but it is not immediately obvious how they relate to each other or how they should be expected to perform in a given environment. In Chapter 4 we categorize several recent exploration methods by the heuristic that the method employs. Our categorization allows researchers to compare the potential of each method and understand their unique contributions to the field.

While discussing the fundamental properties of reinforcement learning can be informative, it is often useful to observe the empirical behaviour of algorithms as well. We present a carefully controlled empirical evaluation of exploration methods in Chapter 5. These experiments are carried out in a set of environments we design to reflect the exploration properties outlined in Chapter 3. Ours is the first systematic study of a diverse set of exploration methods in simple, interpretable environments. The results of our experiments provide additional context to the results presented in the literature, as well as a critical evaluation of our claims elsewhere in this thesis.

In summary, this work studies the environmental factors that make exploration difficult and surveys the different strategies that are used to drive exploration in the literature. We categorize exploration methods according

to their underlying exploration strategies. Finally, we empirically compare a selection of methods from each category on a set of environments that each present a different exploration challenge.

Our proposed contributions are as follows:

1. We identify six distinct properties of environments that make exploration difficult. These properties are derived from the mathematical framework underlying RL problems, and are present in every environment to a different extent. By taking these properties into account, exploration researchers can ensure that their methods address the full scope of RL problems.

2. We design a set of six toy environments that each capture one of the difficult exploration properties. These environments act as a test suite in the sense that a general exploration method should be able to find a near-optimal policy in these small toy environments.

3. We propose a novel categorization of online, model-free exploration methods according to their underlying exploration heuristic. We identify new similarities between methods and discuss how these similarities might affect learning.

4. We present an empirical study of existing exploration methods in the difficult exploration environments described previously. We find that Linear Upper Confidence Least Squares (Kumaraswamy et al. 2018) is able to find a near-optimal policy in all the environments, while other methods find suboptimal policies in at least one environment. We also find that evaluation-phase performance is impacted more by algorithm-level design choices than by the method's exploration heuristic.

The purpose of this thesis is to study incremental, model-free, value-based exploration in reinforcement learning. We begin with a review of the problem setting, where we discuss the reasons that some problems are more difficult

to solve than others. We then survey the exploration literature and categorize methods according to their underlying exploration strategies. Finally, we study the empirical behaviour of each category of methods in environments that pose distinct exploration challenges. By deconstructing the exploration problem and a selection of value-based solution methods, we highlight the importance of developing algorithms that simultaneously address every facet of exploration.

# Chapter 2

# Background & Notation

This chapter begins by introducing the RL setting and the mathematical objects we consider in the rest of the thesis. We then outline two distinct problem settings that both fall under the exploration umbrella. After covering the problem setting, we turn to solution methods. We first describe a simple learning algorithm, then discuss algorithmic considerations that are relevant to our experiments in Chapter 5.

## 2.1 Problem setting

The RL problem setting considers an agent interacting with an environment over a series of discrete time-steps. At some time $t$, the environment is in a certain configuration, or state, $S_t$. We use uppercase variable names to denote random variables, and lowercase names to denote fixed quantities. Depending on the state of the environment, the agent takes an action $A_t$. The environment then generates a next state $S_{t+1}$ using a fixed probabilistic rule that can only depend on the current state $S_t$ and action $A_t$. In addition to the next state $S_{t+1}$, the environment also generates a reward signal $R_{t+1}$ using a fixed rule that can depend on the current state $S_t$, the current action $A_t$, and the next state $S_{t+1}$. This model of agent-environment interaction is known as a Markov Decision Process (MDP).

We restrict our attention to episodic environments, in which the agent's experience is partitioned into episodes. Each episode begins with a state $S_0$

that is drawn from a fixed distribution $p_0$. Episodes end when the agent transitions into a special terminal state, which can be reached from every state by at least one sequence of states and actions.

MDPs can be unrealistic in settings where the complete state of the environment is not available to the agent. For example, consider a gardening robot. Suppose there are not sensors embedded throughout the garden, so the robot can only observe the environment through the sensors on its body. This gardening robot and its environment cannot be described by an MDP. Fortunately, we can easily extend the MDP framework to model the gardening robot.

Suppose that instead of receiving the complete state $S_t$, the agent only receives an observation $\phi(S_t)$ that is a function of state. This model is a generalization of MDPs called a Partially Observable Markov Decision Process. In this thesis we only consider MDPs and POMDPs with a finite number of actions.

A learning agent's goal is typically to maximize the rewards that it accumulates over time. In this formalism, the agent must learn about the environment at the same time that it maximizes reward. Trying to learn quickly while at the same time accumulating reward creates tension between taking actions to explore unknown regions of the environment, and taking the most rewarding actions given the agent's current understanding of the environment. This formalism is therefore called the exploration-exploitation problem after the tension between exploring the environment and exploiting the agent's knowledge.

Here we primarily discuss a simpler setting that lacks this tension, called the pure exploration problem. In this formalism, the agent explores for a finite number of time-steps. After this exploration phase, the agent suggests a fixed behaviour strategy that it believes will accumulate a large amount of reward. The agent is then evaluated by the amount of reward its proposed behaviour accumulates over time.

## 2.2 Solution methods

Agents who solve the pure exploration problem typically learn a probability distribution over actions in each state called a policy $\pi$, which describes the behaviour the agent believes will lead to the largest amount of reward over time. The reward a policy accumulates over time following a state $S_t$ is expressed by the discounted sum of future rewards, called the return:

$$G_t \doteq R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} + \dots$$

Here $\gamma \in [0, 1]$ is a discount rate describing how much the agent cares about rewards in the future relative to the immediate reward.

Since the environment and policy can be stochastic, the return is generally stochastic as well. Pure exploration agents therefore learn a policy that maximizes the expected return, or value, from each state. The value of a policy $\pi$ can easily be expressed in a state-value function, which maps states to expected returns

$$v_\pi(s) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t \sim \pi(s)],$$

or in an action-value function, which maps state-action pairs to expected returns

$$q_\pi(s, a) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a],$$

where the expectation over the $\pi$ implies that actions following $A_t$ are sampled from $\pi$.

### 2.2.1 Algorithms

In this section we discuss the two broad classes of reinforcement learning algorithms which can be used to learn a policy $\pi$ or action-value function $Q_\pi$. The first are value-based algorithms, which learn an action-value function that forms the basis of their behavior. The second are policy-gradient algorithms, which learn a policy parameterized by $\theta$ directly. Policy-gradient

algorithms learn by gradient descent on the objective $J(\theta) = \mathbb{E}_{S_0 \sim p_0}[v_{\pi_\theta}(S_0)]$. In this work we focus on value-based algorithms.

Value-based algorithms learn using individual transitions from a state-action pair to state-action pair. A transition is a tuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$. One of the simplest algorithms, Sarsa, takes its name from the order of the variables in the tuple and in its learning update. At every time-step, Sarsa updates its action-values as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)).$$

These action-value estimates learn the value of the policy $\pi$ that generates the stream of experience.

Value-based agents like Sarsa do not directly learn a policy, so how do they address the pure exploration problem? Consider the action-value function $Q$ learned by Sarsa. At the end of the learning phase, the action-value function $Q$ can be converted into a policy by taking actions with maximal value in each state. This process is called greedification, because the policy $\pi$ acts to greedily maximize the learned values. Greedy policies follow the constraint $\pi(a \mid s) > 0 \iff Q(s, a) = \max_b Q(s, b)$ for all states $s$. Since Sarsa learns action-values for the policy it follows during learning, Sarsa's performance in the pure exploration problem is dependent on the performance of the policy it is following when the learning phase ends.

Agents are not restricted to learning the values of the policy that they follow. In the pure exploration problem, where the agent's goal is to learn an optimal policy $\pi^*$, it may be more appropriate to learn the action-value function corresponding to an optimal policy. We note that optimal policies have values in each state that are at least as high as those of every other policy, and are guaranteed to exist in every MDP (Sutton and Barto 2018). An algorithm that learns the optimal action-value function can accumulate a small amount of reward during the learning phase because it is exploring, but still propose a high-quality policy for the testing phase.

Q-learning (Watkins 1989) is one such algorithm. Its update rule looks

remarkably similar to that of Sarsa, but crucially Q-learning learns the values $q_{\pi^*}$ of the optimal policy $\pi^*$, regardless of the policy that it is following. Sarsa, in contrast, can only learn the values of the optimal policy $\pi^*$ if it is already following it. Q-learning's update rule is as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)).$$

Both Sarsa and Q-learning can be used in POMDPs, where the agent receives an observation $\phi(S_t)$ rather than $S_t$ at each time-step. In this case, the action-values can be approximated by a linear or nonlinear function of the observations $\phi(S_t)$. Many algorithms developed in the last few years use deep neural networks to approximate the value function. While neural networks can express a huge space of functions, and they can be easily trained using gradient descent, these so-called deep RL algorithms have a huge number of hyperparameters whose effects are not well-understood theoretically. An agent can also estimate its action-values using a linear function $Q(\phi(S_t), A_t) \approx \theta^\top \phi(S_t)$ that is parameterized by a weight vector $\theta$. Using a linear function to approximate action-values requires fewer hyperparameters, and can be more easily analyzed theoretically (Melo et al. 2008; Zou et al. 2019).

### 2.2.2 Algorithmic considerations

There are a number of algorithmic adjustments that can be made to Sarsa and Q-learning that change their learning dynamics. We describe a few below to help contextualize the scope of our experiments in Chapter 5.

The Sarsa and Q-learning algorithms both learn from a single, continuous stream of experience that is broken only by the beginning and end of episodes. They only use the data from each transition $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ to do a single learning update, before throwing that data away and moving on to the next transition $(S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2}, A_{t+2})$. We say that an algorithm that learns using one transition at a time is an incremental algorithm. Incremental algorithms contrast with batch algorithms like Least Squares Policy Iteration

(Lagoudakis et al. 2003), which use the entire history of data available to an agent to compute a value function at once.

Learning updates can also be modified to use data from multiple transitions at once. In n-step updates, the value function is updating using multiple time-steps worth of data (De Asis et al. 2018; Hernandez-Garcia et al. 2019). For example, the 3-step Sarsa update is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 Q(S_{t+3}, A_{t+3}) - Q(S_t, A_t)).$$

The learning update can also use a mini-batch of transitions without any specific temporal structure, as in most algorithms that approximate their value functions with neural networks such as DQN (Mnih et al. 2015). We restrict our definition of incremental algorithms to those that use a single transition at a time in their learning updates.

The final algorithmic adjustment we consider is the distinction between so-called model-based and model-free algorithms. The original model-based algorithms used a model of the environment to simulate transitions that the agent would use to update its value function in an incremental manner (Sutton 1991). Later work identified an equivalence between learning from transitions simulated by a model and randomly selected past transitions (van Seijen et al. 2015). We therefore include algorithms that learn from samples of past transitions under the model-based umbrella. Model-free algorithms are those that learn without a model, which typically means that they update the value function from temporally consecutive transitions from the immediate past.

## 2.3 Summary

This chapter described the background necessary to understand the remainder of this work. We introduced the agent-environment interaction central to RL and two mathematical models that represent this interaction, MDPs and POMDPs. We also discussed two of the problem settings within RL: the exploration-exploitation problem in which agents maximize the reward they

receive as well as their learning speed, and the pure exploration problem, in which agents try to learn as quickly as possible without regard to the reward they receive during learning.

We also discuss the objects that agents learn, namely policies and value functions. We present Sarsa and Q-learning, two algorithms that learn value functions using slightly different update equations. Beyond Sarsa and Q-learning, there are several considerations that affect how an algorithm learns. We identify incremental algorithms as those that use a single transition at a time in their learning updates, and model-based algorithms as those that learn from simulated transitions or from samples of past transitions.

# Chapter 3

# Categorizing Environments

From the perspective of any fixed learning agent, some environments are easier to learn than others. When designing algorithms — or even formulating the MDP — it is important to understand what makes an environment more difficult. In particular, it is important to identify the characteristics of MDPs that make them more difficult to explore, and in which the agent is more likely to find a suboptimal policy.

We outline six exploration properties that might cause an agent to learn a sub-optimal policy, and give examples of environments that exemplify these properties. These examples will be more thoroughly discussed in Section 5.1. While the exploration properties we present are not complete, we believe that each presents a distinct challenge to an exploring agent. We also consider extensions to the partially observable setting, where the environment presents an observation rather than a Markov state.

## 3.1 Exploration properties concerning reward

In this section we present three properties of reward that make exploration difficult for value-based RL agents. The reward signal represents how immediately beneficial it is for the agent to take an action in a state. Since an agent takes actions to maximize the discounted sum of rewards, the reward plays a significant role in the agent's decision-making.

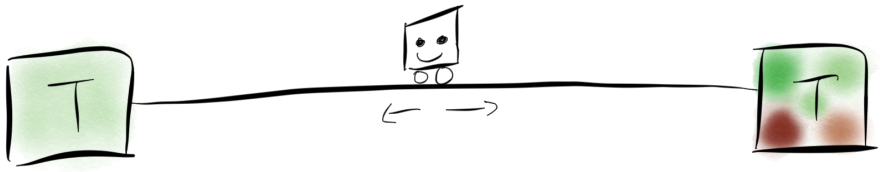

Figure 3.1: Environment diagram for VarianceWorld. The agent starts in the center of the environment and can move left or right. It receives a small, low-variance reward when entering the terminal state on the left, and a large, high-variance reward when entering the terminal state on the right. The environment is about 20 steps across.

### 3.1.1 High variance rewards

Our first exploration property, high variance, simply increases the difficulty of estimating any random variable. As the variance of the reward increases, functions of reward from a state-action pair require more data to estimate accurately. As a result, action-values are more difficult to estimate when rewards are high-variance, and the agent will expect suboptimal actions to be optimal a larger proportion of the time. While this effect can be averaged out over large numbers of runs, we argue that an algorithm with high-variance performance is undesirable to use in practice.

This property is exemplified by VarianceWorld, a continuous navigation environment adapted from White et al. (2010) presented in Figure 3.1. This environment is a short, one-dimensional corridor with a small fixed reward on one side and a large, highly varying reward on the other. The agent begins in the center of the corridor, and episodes terminate when the agent visits either end of the corridor. Despite its simplicity and similarity to a two-armed bandit problem, this environment has proven to be a challenging environment for exploration (White et al. 2010).

### 3.1.2 Misleading rewards

Reinforcement learning agents explore in state-action space by examining local reward signals. If these local signals are misleading, the agent may fail to make globally optimal decisions. Suppose the gradient of the reward with

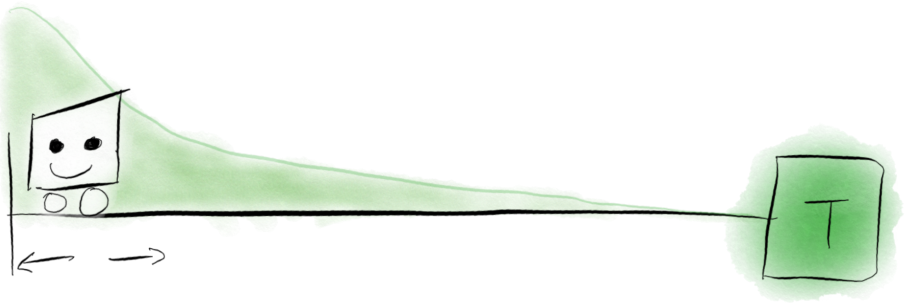


Figure 3.2: Environment diagram for Antishaping. The agent starts at the left of the environment and can move left or right. It receives a small reward, which decreases as the agent moves right. The agent receives a large reward when entering the terminal state on the right. The optimal policy is to move right in every state. The environment is about 200 steps across.

respect to states points in the opposite direction of the gradient of the true action-values in a large region of the state space. Then, as the agent explores, its value function will reflect the gradient of the reward. Depending on how heavily the agent relies on its action-value estimates to select actions, it may fail to explore enough in the direction of greater return, and never visit the most rewarding part of the state space at all.

Consider Antishaping, another one-dimensional corridor environment adapted from Langford (2018) and shown in Figure 3.2. In this environment, the agent starts at the left end of a one-dimensional corridor. As the agent moves right, the reward it receives at each time-step decreases. However, at the right end of the corridor there is a terminal state with high expected reward, so the optimal policy is to always move right. Some agents may be incentivized to focus their exploration on the locally rewarding left side of the environment, and learn a policy with much lower value than the optimal one.

Misleading and high-variance rewards can be combined to create even more difficult environments. For example, consider a modified version of the Antishaping environment in which the agent does not always receive the large reward at the terminal state. Instead, with large probability $1 - \epsilon$, the agent will receive some minimal reward. Depending on $\epsilon$, the agent can spend arbitrarily many time-steps trying to verify that the right end of the corridor

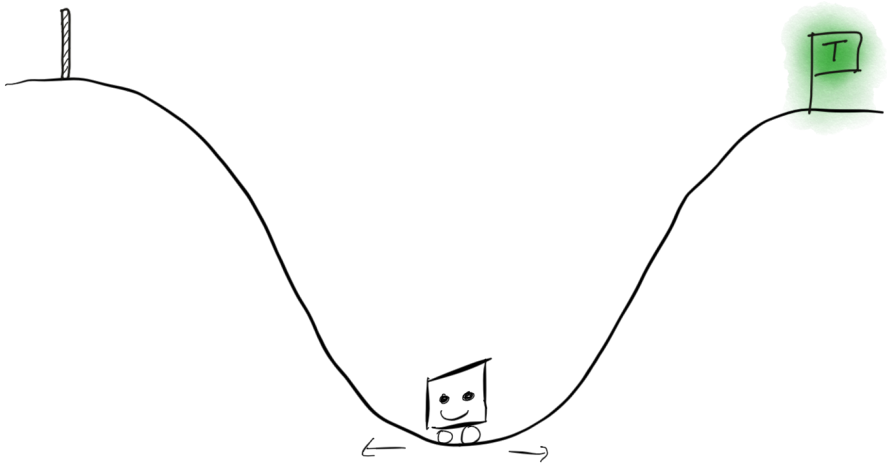

Figure 3.3: Environment diagram for Sparse MountainCar. The agent starts in the valley and can accelerate left or right. It does not have enough force to move up the hill, so it must drive part-way up the left hill and then accelerate towards the right to build up enough momentum to reach the goal. The agent receives a reward when entering the terminal state on the right, and zero reward otherwise. The optimal policy requires roughly 120 time-steps.

has high expected reward. The rewards in this example can be scaled to make the policy of staying at the left end of the corridor arbitrarily worse than staying at the right end.

### 3.1.3 Reward sparsity

When the rewards observed by the agent are all equal to zero, there is no signal in the reward for the agent to optimize. In this case, the agent can only expect all the state-action pairs it has visited to have a value of zero. This zero-everywhere value function implies that all policies are equally rewarding.

The example environment for reward sparsity is a variant of the two-dimensional MountainCar problem (Moore 1990). In this environment, shown in Figure 3.3, the agent must drive an underpowered car up a hill to a goal. Driving uphill requires a sequence of mildly coordinated actions, but even an agent that chooses actions randomly can reach the goal fairly frequently. The original version of MountainCar uses a discount factor of 1 and a reward of -1 at each time-step, which encourages agents to terminate episodes quickly.

As a result, the original version can be solved by a greedy Q-learning agent (Sutton and Barto 2018). Sparse MountainCar uses a discount factor of 0.99, and generates a reward that is positive when transitioning into the terminal state, but zero everywhere else. We discuss the exploration challenges posed by MountainCar and Sparse MountainCar further in Section 3.2.2.

Fortunately, zero reward almost everywhere does not prevent agents from exploring. As we will see in Chapter 4, many exploration methods rely on the heuristic "take actions that the agent has tried less frequently" when rewards are sparse. This heuristic helps agents explore in the following way. Imagine that states are arranged in a graph, with two states connected by an edge if the agent has taken an action in one state to transition to the other. The agent's current state is connected to all other states that it has visited, while unvisited states are not connected to any other nodes. By taking actions that have been tried less frequently, the agent will move towards the boundaries of this graph and eventually visit a new state. The agent can use this heuristic to visit all of the state-action pairs in the MDP.

If the agent plans, as in model-based reinforcement learning, simply visiting each state-action pair enough times to estimate their average rewards is enough to identify the optimal policy. The agent can simply carry out planning updates, which are possible from any state, to learn the optimal action-values. However, exploring the entire MDP is only one of the challenges that reward sparsity presents to model-free agents. Model-free agents must visit a state-action pair to update its value estimate, further increasing the amount of time they need to explore before their action-value estimates reflect the non-zero reward in far-away regions of the state space.

## 3.2 Exploration properties concerning transition dynamics

In this section we present three properties of the transition dynamics that make exploration difficult for value-based RL agents. Since any state-action

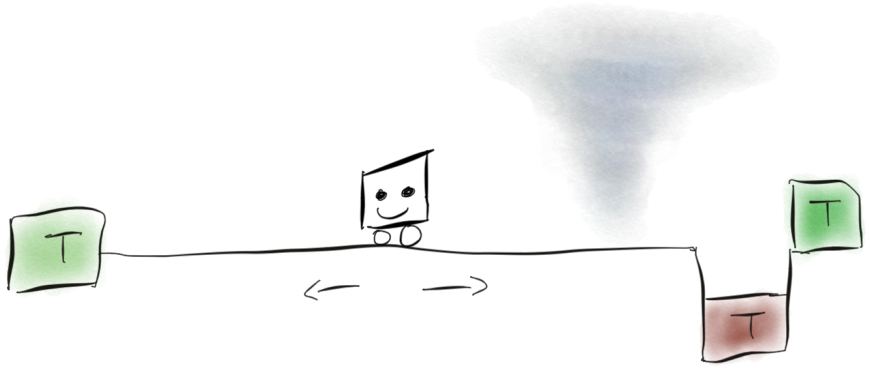

Figure 3.4: Environment diagram for WindyJump. The agent starts in the center of the enviromnent and can move left or right. The agent receives a small reward when entering the terminal state on the left, a large reward when entering the green terminal state on the right, and a large negative reward when entering the red terminal state on the right. Otherwise, the agent receives zero reward. Movement on the right half of the domain is highly stochastic. The environment is about 86 steps across.

pair can have large expected reward, agents must sample the entire state-action space while learning their value functions. Environments in which it is difficult to frequently visit all states can be challenging to explore, especially when the states that are more difficult to visit have high reward. An environment can also be challenging to explore if the states that are likely to be visited cause the agent to believe that suboptimal actions are optimal. Lastly, environments can be difficult to explore if the state-action space is quite large, even if the optimal policy only visits a small fraction of state-action pairs. We describe these properties, with examples, in more detail below.

### 3.2.1 High variance transitions

When transitions are high-variance, the sequence of states visited by an agent are also high-variance. As a result, it can be difficult to estimate the value of a state-action pair without a large number of samples. Consider a state-action pair that might transition to one state with high-probability and another state with low probability. Now suppose the high-probability state has low reward, and the low-probability state has high reward. Similarly to the high-variance

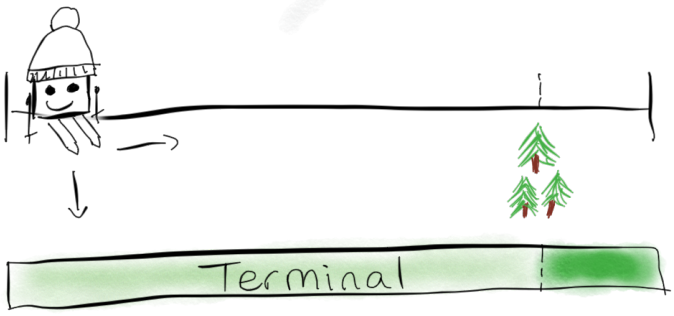


Figure 3.5: Environment diagram for AlpineSki. The agent starts at the top of the mountain, and can ski down to terminate the episode and receive a small reward, or traverse to the right. If it traverses beyond the trees, it can ski down to receive a large reward. The environment is about 20 steps across.

reward case, samples of the return from the original state-action pair are high-variance, and the agent may require many samples to accurately estimate the mean return.

We can modify the high-variance reward domain to have high-variance transitions in the WindyJump domain, shown in Figure 3.4. Again, the left end of the environment contains a small, terminating reward. The right half of the environment now has highly stochastic transitions that may move the agent to any state in a certain region around the agent's current position. At the right end of the environment is a goal region, with a pit immediately left of the goal. If the agent falls into the pit it receives a very small reward, but if the agent reaches the goal region it receives a very large reward. The optimal policy is to move towards the goal region. Due to the stochastic transition dynamics, even the optimal policy has high-variance returns from most state-action pairs.

### 3.2.2 Antagonistic transitions

These environments are difficult to traverse unless the agent takes the correct sequence of actions. In some environments it is easier to imagine that the agent is trapped within a certain set of states unless a sustained effort is made to visit outer states. For example, the RiverSwim environment uses a leftwards-flowing "current" that makes it much easier to move to the left in the MDP than to the right. Since MDPs with antagonistic transitions require more time-steps to traverse, it can take a large amount of time to obtain samples from outer state-action pairs.

The transition dynamics in environments like RiverSwim are not as antagonistic as other domains, such as ComboLock (Langford 2018), DeepSea (Osband, Roy, et al. 2019), or IndexQuery (Du et al. 2019). In these environments, a large reward is given to the agent at a certain goal state or state-action pair. The trick is the goal can only be reached by following a unique sequence of state-action pairs. Since this sequence must be followed exactly to reach the high-value state, stochasticity in the agent's action selection or environment can easily cause the agent's trajectory to deviate from the sequence. The transition dynamics in these environments are antagonistic due to the deterministic connectivity between states in the MDP, rather than gravity in the transition probabilities.

Note that environments with antagonistic transitions do not necessarily have to have sparse rewards. The rewards can have any structure, as long as the policy that most frequently reaches the goal is much better according to the desired evaluation metric. The environments mentioned above all use sparse rewards to make exploration more difficult, and DeepSea even includes a high-variance reward at the goal state.

We give another example of an environment with antagonistic transitions inspired by Combolock called AlpineSki, which is shown in Figure 3.5. In AlpineSki the agent begins each episode at the left side of a mountaintop, which is represented by a one-dimensional corridor. At any time-step, it can

choose to ski down and receive a reward. The agent prefers to ski down at the right side of the mountain, where it receives a large reward. If it skis down anywhere else the agent recieves a small reward. The only way to reach the right side of the environment is to consistently take a 'traverse' action. This action sequence is unlikely to be executed by agents with stochastic policies. In contrast with DeepSea, AlpineSki episodes can last forever, and there is no stochasticity in the reward. As a result AlpineSki presents a simple and focused exploration challenge.

**Transition dynamics in Sparse MountainCar**

Having introduced antagonistic transitions, we can now make a final point concerning the (Sparse) MountainCar environment. MountainCar has a two-dimensional state space, but it is impossible to travel through the state space in a straight line from the start to the goal — instead, the optimal policy traces out a spiral. Intuitively, this description suggests that MountainCar is difficult because of its transition dynamics. However, a Q-learning agent can find the optimal policy in MountainCar without using any exploration method at all (Sutton and Barto 2018). We interpret Q-learning's performance to mean that MountainCar does not possess antagonistic transitions. Since MountainCar does not pose an exploration challenge, the difficulty of exploration in Sparse MountainCar can be attributed to its sparse rewards.

### 3.2.3 Number of states and actions

In the general case, every state and action in an MDP must be explored to identify the optimal policy. Even if an agent uses a function approximator that can generalize over state-action pairs, it may generalize incorrectly if it doesn't explore sufficiently thoroughly. As a result, large state-action spaces force the agent to explore for longer periods of time, even when the optimal policy only visits a small fraction of the state space.

Since any environment can be made trivially more difficult to explore by adding states and actions that are not visited by the optimal policy, our exam-

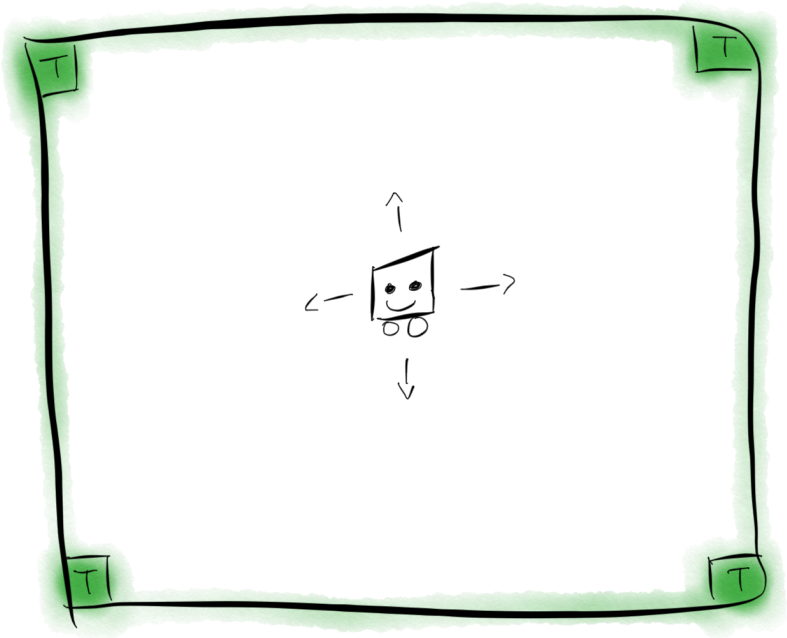

Figure 3.6: Environment diagram for Hypercube. The agent starts in the center of a hypercube and can move in either direction along any axis. To help the agent explore, the reward increases by a factor of around $\frac{1}{1-\gamma}$ for each additional wall the agent touches. When the agent touches a corner of the hypercube, the episode terminates with a large reward. The hypercube has a radius of roughly 10 steps.

ple environment demonstrates a different effect. We propose the Hypercube environment, which is a small n-dimensional cube shown in Figure 3.6. The agent can move in either direction along any of the axes, with the goal of reaching one of the corners of the hypercube, where the episode terminates. To help the agent reach a corner, a shaping reward is provided to the agent that depends on the number of walls it is touching. While the number of states and actions both increase exponentially with the number of dimensions, the symmetry in the environment means that the number of corners, or goal states, increases exponentially as well. This symmetry tempers the difficulty of exploration, especially if the agent can generalize accordingly.

## 3.3 Combining difficult rewards and transition dynamics

Transition dynamics can have an even greater effect when combined with misleading rewards — we consider the simple example of RiverSwim (Strehl et al. 2008). In RiverSwim, a current pushes the agent to the left, where at the end of the river the agent can continuously receive a small reward. However, if the agent swims upstream, to the right, it will eventually find a state where it continuously receives a large reward. Between the small downstream reward and the relative difficulty of swimming upstream, some agents never explore the high-value state at the source of the river.

## 3.4 Impact of the state representation

The exploration problem changes slightly when the the agent only sees a partial observation $\phi(S_t)$ rather than the complete state $S_t$. The above properties will generally still apply in partially observed settings, especially those based on modifying the reward. Due to the close relationship between the state representation and the state transition dynamics, the exploration properties related to the transition dynamics should be interpreted in the context of the agent's function approximation. For example, if the agent's function approx-

imation generalizes appropriately, it may be able to learn more quickly in an environment with a large state-action space by sharing weights between similar state-action pairs. However, if the function approximation generalizes poorly, those same shared weights will induce incorrect beliefs, and the agent will spend an extended period of time exploring.

How the state representation interacts with exploration is an active area of research. Quite recently, Lattimore et al. (2019) showed that near-optimal policies can be identified in time polynomial in the function approximation's error, number of features, and horizon $\gamma$. The runtime of their algorithm notably does not depend on the size of the state-action space. This result relies on features that are able to express the optimal value function fairly closely, including for example generalizing between similar state-action pairs, as discussed previously. For further discussion see Du et al. (2019) or Van Roy et al. (2019). While reinforcement learning may be efficient with the appropriate choice of features, it remains to be seen whether those features can be learned efficiently during interaction with the environment.

## 3.5 Summary

Reinforcement learning problems can be made more difficult by modifying the distribution of rewards or state-action transitions. Variance in rewards and state-action transitions can increase the number of samples necessary to estimate the optimal value function. If an agent's behaviour relies too heavily on its value estimates in early learning, misleading rewards can cause the agent to stop exploring prematurely. Antagonistic transitions can have a similar effect. Alternatively, they can greatly extend the exploration time if the agent is committed to a minimum number of samples from each state-action pair. Sparse rewards lack information that may help guide exploration in early learning, which can be exacerbated by large state-action spaces. Model-free algorithms also take longer to propagate information through environments with large state-action spaces since they only update the value of state-action

pairs when visiting them.

Partial observability adds another layer of complexity to the reinforcement learning problem, but not all is lost. Agents in this regime estimate the value function using a function approximator, whose generalization properties may help the agent avoid sampling every state-action pair. Several authors (Lattimore et al. 2019; Van Roy et al. 2019) have recently identified characteristics of feature representations that are necessary for agents to learn near-optimal policies in polynomial time. It remains to be seen whether it is possible to learn such features efficiently in an online and incremental manner.

# Chapter 4

# Exploration methods

Now that we have seen how environments can challenge a reinforcement learning agent, it is time to review the literature and see what kinds of exploration methods have been proposed. To focus the thesis and our empirical study, we restrict our attention to value-based agents that are incremental and model-free. Despite these restrictions, a large body of exploration research remains within our scope.

Notably, we omit methods that behave according to a softmax (Duncan 1959) or Boltzmann distribution over the action-values. This strategy is typically used with policy gradient methods (Degris et al. 2012), or in entropy regularized reinforcement learning (Asadi et al. 2017; Azar et al. 2012; Haarnoja et al. 2017; Kozuno et al. 2019; Nachum et al. 2017; Peters et al. 2010; Still 2009), which effectively changes the MDP being solved. Neither of these settings are within our scope.

There are three main approaches that drive exploration methods. The first approach, *optimism*, takes actions if they are part of a policy that might be optimal. The second approach measures the agent's *learning progress* according to the error in some prediction, and encourages the agent to revisit states where it can learn more quickly. The third approach behaves greedily according to the agent's value function, but adds noise so that exploratory actions are chosen by chance.

Optimistic methods choose actions greedily with respect to a plausible model of the environment, often represented by a value function. They de-

termine whether an action could be optimal using one of two strategies: Optimism in the Face of Uncertainty (OFU), or Thompson Sampling. takes the action with the highest value according to some index. The index is frequently an upper confidence bound on the optimal action-value function, or some proxy thereof. At each time-step, the action with the highest index, or highest potential value, is chosen in each state. This strategy is called Optimism in the Face of Uncertainty because the agent behaves as if this high potential value can be realized.

What does OFU have to do with exploration? Well, an upper confidence bound (or its proxy) might be high for two reasons. It could be a tight bound, in which case the action is known to have a high value. It could also be a loose bound, in which case not much is known about the action. The exploration problem asks when it is appropriate to exploit known high-value actions and when it is necessary to explore lesser-known actions further. By identifying a measure that expresses uncertainty in the same units as value, OFU can leverage existing statistical tools to efficiently test the most relevant actions.

While OFU methods are deterministic, our next strategy is inherently stochastic. This strategy maintains a distribution over value functions that estimates how probable it is that any given value function is optimal. The agent samples a value function from the distribution and behaves greedily, or optimistically, with respect to the sampled function. The experience it gains is then used to update the distribution. This strategy is called Thompson Sampling (TS) after Thompson (1933), who applied it to a simple bandit. In its original Bayesian formulation, the probability mass will eventually be concentrated over the optimal value function. However, some recent implementations do not update the distribution in a Bayesian fashion. In problems where it is difficult to design a confidence interval that is both statistically and computationally efficient, Thompson Sampling often outperforms OFU methods (D. J. Russo et al. 2018).

Exploration methods must ensure that states are sampled sufficiently often to learn a high-value policy. But how can the agent know when it has enough

samples? Our next strategy uses the learning progress of a secondary function to chart how optimistic the agent should be. Suppose that in addition to learning action-values, the agent also estimates the output of an arbitrary function over the agent's feature space. Then, the error in this secondary prediction approximates how well the agent has learned the value function in the corresponding region of the state space. A high secondary error suggests that the agent may not have observed enough samples from that region of the state-action space, and is converted into a reward or value bonus.

The Learning Progress methods we survey all behave greedily with respect to value estimates that are inflated by a bonus. Despite their use of inflated value functions, we claim that Learning Progress methods are conceptually distinct from Optimistic methods. Optimistic methods are optimistic in a statistical sense; their inflated value estimates are inflated to form an upper confidence bound on the true value estimates. Their use of confidence bounds means that Optimistic value bonuses decay polynomially in the number of samples that the agent observes. In learning progress methods, value bonuses decay as quickly as the auxiliary function is learned. Since most of the surveyed methods learn their auxiliary functions using gradient descent, the auxiliary function's error decays exponentially in the number of samples seen by the agent. These different decay rates are the primary reason we do not believe Learning Progress methods are rough approximations of OFU methods.

The third approach to exploration is to add noise to the value function or action-selection mechanism. This noise can be fixed or adapt according to the agent's experience. While adding noise is a heuristic, these methods have enjoyed a long history of empirical success, and are among the most popular exploration methods used today (M. Bellemare et al. 2016).

## 4.1 Optimistic methods

In this section we describe the properties of Optimistic exploration methods, including OFU and Thompson sampling methods. These methods behave greedily, or optimistically, with respect to a statistically plausible value function. The restriction to statistically plausible value functions typically means that these methods' optimism decays with $\frac{1}{\sqrt{(n)}}$ in the number of samples as it would decay in a confidence bound. The rigorous statistical theory underlying optimistic methods has resulted in strong performance across many implementations and domains.

### 4.1.1 Optimism in the Face of Uncertainty

There is no shortage of OFU exploration methods. This line of study focuses on maintaining upper confidence bounds over action values, and taking the action with the highest upper bound on each step. These methods are based on the famous Upper Confidence Bound algorithm (T. L. Lai et al. 1985) for multi-armed bandits. OFU is typically used in RL by adding a per-step uncertainty estimate to the reward during the learning update. This update creates an approximate confidence interval around the value function, and was introduced to RL through the Interval Estimation Q-learning (IEQL+) algorithm (MEULEAU 1999). IEQL+ inspired a model-based algorithm with strong theoretical guarantees called Model-Based Interval Estimation—Exploration Bonus (MBIE-EB) (Strehl et al. 2008) which is the foundation of most recent model-free OFU methods.

The first extension of MBIE-EB to the non-linear function approximation setting combined Deep Q Networks (DQN) (Mnih et al. 2015) with MBIE-EB's approximate confidence intervals (M. Bellemare et al. 2016). These intervals were computed with approximate state-visitation counts, computed with the help of a separate network that estimated the probability of visiting each state. This work inspired a wave of papers that extend the method by counting state visitation counts in different ways.

One method hashes observations into clusters, and counts the frequency of visitation to each cluster (Tang et al. 2017). Other methods use the agent's existing feature representation to estimate state visitation: $\phi$-pseudocounts approximates state visitation frequency based on the distribution of activations of the agent's features (Martin et al. 2017).

More recent methods use value functions of different rewards to count state visitation. DORA (Choshen et al. 2018) predicts a single constant-zero value function that is initialized to one everywhere, then use a log-transform of the values to approximate state visitation. State visitation can also be approximated by the L1 norm of a vector called the successor representation (Machado, M. G. Bellemare, et al. 2018). The successor representation (Dayan 1993) contains one value function for each feature, where the reward is the magnitude of that feature on each time-step.

However the agent counts how often it has visited a state or state-action pair, the above methods all encourage exploration by adding a multiple of $\frac{1}{n(s)^2}$ to the agent's reward, where $n(s)$ is the number of visits to state $s$.

While most of the OFU methods we survey use a reward bonus to approximate an upper bound on the value function, upper bounds can also be estimated directly. Kumaraswamy et al. (2018) propose a method called Linear Upper Confidence Bound Least Squares (L-UCLS), which estimates the variance of the value function using a second set of weights. The variance estimate is then used to construct an upper confidence bound on the values. At each time-step, the agent takes the action that has the highest upper confidence bound.

### 4.1.2 Thompson Sampling

Thompson Sampling (TS) requires the agent to sample a solution from a distribution and test its performance. The following methods each represent a distribution over value functions and behave greedily with respect to samples from the distribution. We note that L-UCLS's variance estimates can be used to construct an OFU method that behaves greedily with respect to upper con-

fidence bounds, or a TS method which samples from a distribution over value functions on each time-step.

Gal et al. (2016) represent their value function distribution with a single neural network. The authors show that a popular method to prevent overfitting called dropout (Srivastava et al. 2014) can enable the network to represent a distribution. Dropout randomly excludes nodes from each forward pass, allowing the network to generate samples. This method can be easily extended to any reinforcement learning agent that uses a neural network to represent the value function.

Alternative techniques such as the statistical bootstrap can also be used to maintain and sample from a distribution over value functions. Bootstrap DQN (Osband, Blundell, et al. 2016) uses a set of neural networks or other function approximator to construct a bootstrap distribution. Bootstrap DQN explores by choosing actions greedily with respect to a randomly selected value function. The authors stress the importance of using the same bootstrap sample for the duration of an episode, which allows the agent to coordinate its behaviour over multiple time-steps.

The randomization in a bootstrap sample that drives exploration exclusively comes from data from the environment. Early exploration can be improved by adding an initial source of randomness, as in Bootstrap DQN with Randomized Prior Functions (Osband, Aslanides, et al. 2018). These randomized prior functions also motivate the agent to explore in the absence of environmental stochasticity.

## 4.2 Learning progress methods

In addition to learning an action-value function, learning progress agents learn an secondary function over a state-action feature space. The errors in predictions of the secondary function act as a proxy for errors in the value function, providing a signal roughly describing how well the value function has been learned. Note that the secondary function does not need to mir-

ror the value function by mapping features to real numbers. For example, some methods learn models of the environment dynamics with their secondary functions. However, these methods are still considered "model-free" because they do not update the value function using transitions generated by the model.

When the agent receives observation signals rather than state, it is often useful to preprocess the observations into a more compact space before making a prediction. Stadie et al. (2015) begin by learning a compact embedding of the observation using an autoencoder. The current observation's embedding is used to predict the code of the next time-step's observation. The corresponding error is passed to the RL agent as a reward bonus. Since this method predicts embeddings of observations, this method is most appropriate when every observation contains enough information to create a Markov state.

The observation code learned by an autoencoder may capture information that is irrelevant to the agent's policy. A different preprocessing approach is to use the features learned by a model that predicts the current time-step's action given the current and next observations (Pathak et al. 2017). At each time-step, the current action and features are used to predict the next time-step's features. Then, the prediction error is passed to the agent as a reward bonus. In this method, the model only needs to learn features of the observations that cause or are caused by actions. In particular, there is no incentive for it to learn features of the environment that are not controllable by the agent. This differs from an observation embedding learned by an autoencoder, which might learn any number of irrelevant features, unnecessarily increasing the difficulty of the resulting prediction problem.

In some situations the prediction error may not behave as desired to guide exploration. For example, a state that generates a random observation will always result in a high prediction error, even after the agent observes many samples. This affinity for stochastic observations can be avoided if the agent's secondary function predicts a deterministic target rather than a stochastic tar-

get like the next observation. In Random Network Distillation (Burda et al. 2018), the secondary function $\hat{f}_w(S_t)$ with weights $w$ predicts a fixed, randomly initialized target $f(S_t)$. Since the target function is deterministic, the agent will not receive large reward bonuses for visiting stochastic areas of the environment. However, as desired, the prediction error will decrease as the agent observes more samples each region of the state space.

A common theme in learning progress agents is that their reward bonuses decay at the rate that the agent learns its secondary function. Since most methods learn using a variant of stochastic gradient descent, the learning rate is geometric in the number of samples. This learning rate is preserved in a method called Optimistic Initialization, which implements exploration by decreasing the agent's action-value estimates.

### 4.2.1 Optimistic Initialization

The final Learning Progress method is Optimistic Initialization, which explores by acting greedily with respect to an initially inflated value function. We describe it in detail and discuss our reasons for categorizing it as a Learning Progress method rather than an Optimistic method.

When an optimistically initialized agent first takes an action $a$ from a state $s$, it discovers that the resulting reward is not nearly as high as the agent expects. To compensate, the agent decreases its action-value estimate $Q(s, a)$ for the action that it took. Since $Q(s, a)$ is now lower than action-values for unexplored actions, the agent will choose a different action next time it revisits that area of the state space.

At first glance, Optimistic Initialization does not appear to be a Learning Progress method, or even a proper exploration method at all. However, it has the characteristics of a Learning Progress method: it behaves greedily with respect to an inflated value function, the value function is inflated by the prediction error of a secondary function, and the error decays geometrically in the number of samples observed by the agent. It is easy to see that Optimistic Initialization behaves greedily with respect to inflated values, but it is harder

to see what acts as the secondary function.

The secondary function can be found by considering the difference between the value estimates learned by an optimistically initialized agent $Q_t^{\text{opt}}(s, a)$ and by an agent with a different initialization scheme $Q_t(s, a)$. Call the difference

$$Q_t^+(s, a) = Q_t^{\text{opt}}(s, a) - Q_t(s, a). \tag{4.1}$$

In most RL algorithms such as Sarsa, there is no difference between learning the combined value function $Q_t^{\text{opt}}$ with a reward of $r^{\text{opt}} = r + 0$, or learning the value functions $Q_t$ and $Q_t^+$ separately with with rewards of $r$ and 0 respectively, where $r$ is the reward signal from the environment. Not only will both processes produce the same final value function $Q^{\text{opt}}(s, a) = Q(s, a) = q(s, a)$, but equation 4.1 will also hold during learning, as long as both processes learn from the same trajectory of experience.

Notice that the secondary function $Q_t^+$ has a target that is 0 everywhere. Consequently, its error at a state-action pair $(s, a)$ is simply $Q_t^+(s, a)$ itself. So Optimistic Initialization is a Learning Progress method that uses the error in its secondary function $Q_t^+(s, a) - 0$ as a value bonus to inflate its action-value estimates $Q_t(s, a)$.

Astute readers may notice that equation 4.1 does not hold in general when Q-learning is used, because the same action might not maximize the value of the next state for both $Q_t$ and $Q_t^+$. On those time-steps, $Q_t^{\text{opt}} \leq Q_t + Q_t^+$ due to Jensen's inequality. Fortunately, as long as the agent behaves greedily with respect to the combined action values, the Q-learning update and the Sarsa update become identical, so equation 4.1 will always hold. The underestimation due to Jensen's inequality only appears when optimistic initialization is combined with another exploration technique like epsilon-greedy.

Optimistic Initialization is implemented slightly differently depending on the function approximation used by the agent. If the agent's value function is estimated with a linear function of weights or in a table, over-estimation can be ensured by simply setting the initial weights much higher than the designer expects they should be in the optimal value function. However, methods that

update weights globally, like neural networks, typically do not maintain initial overestimation during learning. Optimistic Initialization can still be used in neural networks by decreasing the rewards instead of increasing the initial values (Machado, Srinivasan, et al. 2015).

## 4.3 Noisy methods

These methods are among the oldest exploration heuristics to be used in reinforcement learning. They are also the only methods we survey that randomize action-selection on every time-step, falling into the category of undirected exploration (Kumaraswamy et al. 2018; Thrun 1992). These methods can take exponentially many time-steps to exploring environments with antagonistic transitions (Osband, Roy, et al. 2019). Despite their theoretical shortcomings, these methods are simple and are the most commonly used in practice.

Epsilon-greedy is an incredibly popular strategy that randomizes the agent's choice of actions. It usually selects the action with the highest estimated value, but with probability epsilon it selects a random action. Since the exploration is independent across time-steps, it is difficult for the agent to visit distant states that would not otherwise be visited by the greedy policy. Despite this limitation, epsilon-greedy exploration is sufficient for Q-learning to converge to the optimal policy in tabular MDPs (Melo 2001; Tsitsiklis 1994), and frequently improves performance when using function approximation as well.

Rather than randomizing the policy, the agent can randomize the action-value function as well. This is the idea behind Noisy Networks (Fortunato et al. 2018), which parameterizes the value function with mean and noise parameters. To calculate the values, the noise parameters are multiplied by a sample from a fixed, zero-mean noise distribution and added to the mean parameters. The noise parameters are updated using gradient descent in the same manner as the mean estimates, creating a simple method whose exploration is tuned automatically as the agent learns. A similar method is

presented by Plappert et al. (2018).

## 4.4 Summary

We have reviewed a wide range of methods that have been used to explore in incremental, online, and model-free reinforcement learning. There are three main categories of methods. Optimistic methods, which can be further divided into those based on OFU and Thompson Sampling, take actions that could plausibly have high value. Learning progress methods adjust the actions the agent takes based on the learning progress of another function. Finally, Noisy methods randomize the agent's action-selection process to explore around the agent's estimate of the optimal policy. In Chapter 5 we outline a set of experiments to better understand the empirical performance of these exploration methods.

# Chapter 5

# Experimental setup

We examine a selection of exploration methods and evaluate their performance on a series of simple environments. The selection consists of representative methods from each category defined earlier in Chapter 4. To keep the experiments practical we restrict our scope to incremental, model-free methods whose computation scales linearly in the number of features every time-step. These methods only use the data from each transition once to update the value function, then throw the data away.

Each exploration method is adapted to work with a Q-learning agent with tile-coded linear function approximation augmented by a bias feature. Our environments are modeled after the properties described in Chapter 3 They are each designed to present a single challenging facet of the exploration problem in reinforcement learning.

Since this is the first systematic empirical study of value-based exploration in RL, we consider the most fundamental exploration setting. This setting is pure exploration, in which agents explore for a certain amount of time and then output a near-optimal policy. Pure exploration provides a simple, concrete evaluation metric — the performance of the final output policy. Performance is harder to measure in the exploration-exploitation tradeoff because both the reward accumulated during learning and the quality of the final policy must be accounted for.

The following experiments are split into a training phase and a testing phase. The training phase lasts 500000 time-steps, which is more than suffi-

cient to learn the optimal policy in each of these environments. In the testing phase, the agent's learning and exploration mechanisms are disabled, and performance of the greedy policy with respect to the agent's value function is evaluated for 100000 time-steps. The testing phase is long so that there is time to average over stochasticity in the environment or starting conditions. Episodes are not cut off for any reason unless the budgeted number of time-steps is exceeded.

Each hyperparameter is swept over a group of candidates that were chosen based on reported values from the papers that originally proposed these exploration methods. They are appropriately diverse, since most methods performed best with different settings on each environment, and rarely performed best with the most extreme parameter settings.

We chose hyperparameters, including function approximation settings, separately for each environment. Since performance tended to be distributed bimodally, we identified the best hyperparameters by their median cumulative reward during the testing phase. Each hyperparameter setting was run 72 times to ensure that the median was representative of the underlying distribution over performance.

It is unclear whether there is a sensible way to disable exploration during the testing phase for some of the tested methods. Both the ensemble learned by BootstrapQ and the random value function learned by NoisyNetwork cannot easily be determinized, so we opted to evaluate the frozen ensemble and random value function while keeping their exploration strategies intact. SoftmaxAC also learns an inherently stochastic policy, so we allowed the policy to be stochastic during the testing phase.

## 5.1 Environments

The following environments are designed to be as simple as possible. They have continuous state, which contributes to their partial observability, and simple finite action sets. In these experiments we use a discount factor of

$\gamma = 0.99$. Each environment is structured so that the value of the optimal policy from the start state is roughly 1. In some of the environments, agents who do not explore enough will tend to learn a specific suboptimal policy. These suboptimal policies have a value around 0.01 from the start state.

### 5.1.1 High-variance reward: VarianceWorld

This environment is adapted from the noisy reward navigation task by White et al. (2010). In many ways it is similar to a multi-armed bandit. The agent starts near the center of a one-dimensional corridor, roughly 10 steps away from either edge. The state is a corridor, represented by $(0, 1)$. The agent starts uniformly in state $[0.45, 0.55]$, and has two actions that move the agent to the left or the right according to a $\mathcal{N}(\frac{1}{20}, 10^{-4})$ distribution. Episodes terminate when the agent moves past $s = 0$, in which case the agent receives 0.011 reward, or past $s = 1$, in which case the agent receives reward drawn uniformly from $10, -10, 1, -1, 5.5$. As promised, the 'always go left' policy has a value near 0.01, and the 'always go right' policy has a value near 1.

### 5.1.2 Misleading reward: Antishaping

This environment was adapted from Langford's (2018) Antishaping environment. It is also a corridor with state in $(0, 1)$, and the agent starts uniformly in $[0.45, 0.55]$. In this environment it takes 100 steps to reach the edge, as agents can move $\mathcal{N}(\frac{1}{200}, 10^{-4})$ to the left or the right. The reward signal in Antishaping is designed so that the policy that moves to $s = 0.5$ and remains there forever has a discounted value of about 0.01, as measured from the start state distribution. For this implementation, the non-terminating reward in state $s$ is $\frac{P_{\mathcal{N}}(0.5-s)/0.15}{3989}$, where $P_{\mathcal{N}}$ is the probability density function of a standard normal distribution. Antishaping also provides a reward of 2.73 when the agent terminates the episode by exiting the boundaries of the corridor. This terminal reward was chosen so that the expected start state's value under the policy that always moves towards the nearest end of the corridor is roughly 1.

### 5.1.3 Sparse reward: Sparse MountainCar

The MountainCar environment (Moore 1990) is a classic toy problem in RL. The agent is an underpowered car that uses momentum to drive to the top of a hill, where the episode terminates. The reward is typically $-1$ per time-step, incentivizing discounted agents to terminate the episode as quickly as possible. In our implementation, the reward is 0 everywhere, with 3.34 reward upon termination. This reward was chosen so that the optimal policy has value close to 1 in the start state distribution. MountainCar's dynamics are defined by the position $x_t$, velocity $\dot{x}_t$, and action $A_t \in \{-1, 0, 1\}$ of the agent-car. The following state update rules are copied from Sutton and Barto (2018) for the reader's convenience.

$$x_{t+1} = \text{bound}\,[x_t + \dot{x}_{t+1}]$$
$$\dot{x}_{t+1} = \text{bound}\,[\dot{x}_t + 0.001A_t - 0.0025\cos(3x_t)]$$

The bound operator keeps the position within $[-1.2, 0.5]$ and the velocity within $[-0.07, 0.07]$. If the dynamics update causes the agent's position $x_t + 1$ to go beyond $-1.2$, the velocity $\dot{x}_{t+1}$ is set to 0.

### 5.1.4 High-variance transitions: WindyJump

The WindyJump environment is similar in spirit to the VarianceWorld environment, this time creating stochasticity in samples of the return by randomizing the state transition dynamics rather than the rewards. Episodes begin with the agent standing in a corridor represented by $[0, 1]$, in a state drawn uniformly from $(0.45, 0.55)$. The agent's goal is to jump over a pit on the right side of the environment to reach a large reward of 9.5. The location of the pit corresponds to $(1, 1.06]$. Whether the agent jumps over the pit or falls into it, the episode terminates when the agent leaves the $[0, 1]$ interval. There is also small reward of $0.01\gamma^{-43}$ at the left end of the corridor.

WindyJump has two actions, which behave quite consistently in the left half of the environment where there is no wind, and quite stochastically on the

right half of the environment where there is wind. In the wind-free region, the agent can move left or right according to a $\mathcal{N}(0.0116, 10^{-4})$ distribution. On the windy right half of the environment, the agent's movement is much more stochastic, going left or right according to a $\mathcal{U}(-0.1, 0.125)$ distribution. The value of the policy that always takes the left action is roughly 0.01, while the value of the policy that always takes the right action is roughly 1.

### 5.1.5 Antagonistic transitions: AlpineSki

AlpineSki is a simple combinatorial search problem inspired by Combolock (Langford 2018) and a highway commute environment (D. Russo 2019). The agent is an alpine skier moving downhill towards the bottom of the mountain, where the episode terminates. At each time-step, the skier can choose to ski down, terminating the episode, or to traverse across the top of the mountain in search of a steeper slope. We represent the top of the mountain with a state $s \in [0, 1]$, where the skier starts at $s = 0$. The traverse action moves the agent to the right according to a $\mathcal{N}(\frac{1}{20}, 10^{-4})$ distribution. The skier prefers to ski down sufficiently steep slopes, which can be found by taking the ski action in states $s \in [0.95, 1]$. Skiing down a steep slope results in a reward of $\gamma^{-20}$, whereas skiing down another slope results in a reward of 0.01. Like the other environments, the value of the optimal policy from the start state is roughly 1, and the value of skiing down immediately is 0.01.

### 5.1.6 Large state-action space: Hypercube

The Hypercube environment has a very large 3-dimensional state-space. As you may have guessed, the environment is a hypercube represented by $s \in [-10, 10]^k$. The agent starts each episode at the origin, and can move in either direction along any of the axes by a distance sampled from $\mathcal{N}(1, 0.15^2)$. The objective of the agent is to travel to the corner of the hypercube, where the episode will terminate. Since the state-action space is so large, we add shaping rewards to help the agent find the optimal policy. The agent receives increased reward as it touches more walls of the hypercube. The increase in

reward when the agent touches another wall is such that the reward from one time-step of touching 2 walls is larger than the discounted return from touching just one wall forever. The rewards are $7.3 \times 10^{-7}$, $8.1 \times 10^{-5}$, $9 \times 10^{-3}$, and 1.245}. They are scaled so that the optimal policy has a value near 1 at the origin.

## 5.2 Exploration methods included in the study

We test variations of each representative method where applicable. For example, reward bonuses based on state visitation can also be computed from state-action visitation. We also test three different settings of tile-coding to empirically study the relationship between an algorithm's features and how it explores during learning.

Tile-coding differs from neural networks, the typical non-linear function approximation used in reinforcement learning, in two major ways. First, neural networks must be augmented in several ways to become stable enough to use with reinforcement learning (Mnih et al. 2015). These augmentations interact with reinforcement learning in ways that are not yet fully understood, as well as adding several hyperparameters that must be appropriately tuned. As a result, using linear function approximation is both more interpretable and less computationally taxing than using neural networks for reinforcement learning.

Second, tile-coded features are fixed over time. If a tile-coded feature is activate, it is always caused by the same phenomenon in the agent's observation. The closest analogue to linear features in neural networks is the nodes in the last layer of the network. However, neural networks typically learn features concurrently with the value function, so the same node in the last layer may be activated by different environmental phenomenon at different time-steps during learning. In a general exploration context, an agent would have to learn a mapping from some compression of its experience to expected returns. Unfortunately, this is an open problem and remains an interesting

direction for future work. In these experiments we restrict our attention to exploration with linear function approximation, which remains a vibrant area of research (Du et al. 2019; Lattimore et al. 2019; Van Roy et al. 2019).

We use three variants of tilecoding to ensure that the results are robust to different feature settings. The first uses two tiles and 32 tilings, which allows the agent to generalize significantly, but prevents it from discriminating finely between nearby states in the early stages of learning. The second variant uses 8 tiles and 8 tilings, while the third uses 32 tiles and 2 tilings. This last variant has a more local function approximation, so that values learned in one area of the state space does not generalize to other regions. All variants include a bias weight whose associated feature is always 1. While not representative of all linear function approximation schemes, these representations differ greatly in the ways they generalize over the state space and discriminate between values of nearby states.

We set the learning rate $\alpha = \frac{\alpha_0}{\#\text{tiles}}$ for each algorithm and sweep over $\alpha_0 \in \{1.0, 0.5, 0.25, 0.125, 0.0625, 0.03125\}$. Unless otherwise specified we initialize all weights to zero at the beginning of the training phase.

### 5.2.1 Optimism in the face of uncertainty

OFU methods provide a reward bonus to the agent based on the number of visits to the current state. We can approximate these counts by estimating a feature visitation density function, as in $\phi$-pseudocounts (Martin et al. 2017). This method constructs a feature visitation density that is the product of independent visitation densities for each element of the feature vector. The density for the $i$th feature at time $t$ is

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

where $N_t(\phi_i)$ is the number of activations of $\phi_i$ up to and including time $t$. The density over the whole feature space is calculated as a product over all individual feature densities $\rho_t(\phi) = \prod_i \rho_t^i(\phi_i)$. Now let $\rho_t(\phi)$ be the density after observing $\phi_\tau$ up to time $t$, and $\rho'_t(\phi)$ be the density after observing $\phi_\tau$ up

**Algorithm 1:** Q-learning with state-action count-based reward bonuses

**Input:** arbitrary value function weights $\mathbf{w} \in \mathbb{R}^d$
**Input:** discount factor $\gamma \in [0, 1]$
**Parameters:** learning rate $\alpha > 0$, reward scaling parameter $\beta > 0$,
feature map $\boldsymbol{\phi} \colon S \times A \to \mathbb{R}^d$
**Output:** action-value function $Q(S, A) = \mathbf{w}^\top \boldsymbol{\phi}(S, A)$

**until** *training budget exhausted* **do**
  $S \leftarrow$ initial state of the episode
  $A \leftarrow \operatorname{argmax}_a \mathbf{w}^\top \boldsymbol{\phi}(S, a)$
  **until** *episode terminates* **do**
    Take action $A$, observe $R$, $S'$
    $A' \leftarrow \operatorname{argmax}_a \mathbf{w}^\top \boldsymbol{\phi}(S', a)$
    Update visitation count estimates $\hat{N}(S, A)$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[ R + \frac{\beta}{\hat{N}(S,A)} + \gamma \max_a \mathbf{w}^\top \boldsymbol{\phi}(S', a) - \mathbf{w}^\top \boldsymbol{\phi}(S, A) \right] \boldsymbol{\phi}(S, A)$
    $S \leftarrow S'$, $A \leftarrow A'$
    Exit if *training budget exhausted*
  **end**
  Update visitation count estimates $\hat{N}(S, A)$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[ R + \frac{\beta}{\hat{N}(S,A)} + \gamma \max_a \mathbf{w}^\top \boldsymbol{\phi}(S', a) - \mathbf{w}^\top \boldsymbol{\phi}(S, A) \right] \boldsymbol{\phi}(S, A)$
**end**

**Algorithm 2:** IEQL+ for estimating $Q \approx q^*$

**Input:** arbitrary value function weights $\mathbf{w} \in \mathbb{R}^d$
**Input:** discount factor $\gamma \in [0, 1]$
**Parameters:** learning rate $\alpha > 0$, reward scaling parameter $\beta > 0$, standard deviation of return $\sigma_{\max}$, confidence interval width $1 - \theta$, feature map $\boldsymbol{\phi} \colon S \times A \to \mathbb{R}^d$
**Output:** action-value function $Q(S, A) = \mathbf{w}^\top \boldsymbol{\phi}(S, A)$

Initialize $Q(S, A) = \frac{\sigma_{\max} Z_{\theta/2}}{1-\gamma}$
**until** *training budget exhausted* **do**
  $S \leftarrow$ initial state of the episode
  $A \leftarrow \operatorname{argmax}_a \mathbf{w}^\top \boldsymbol{\phi}(S, a)$
  **until** *episode terminates* **do**
    Take action $A$, observe $R$, $S'$
    $A' \leftarrow \operatorname{argmax}_a \mathbf{w}^\top \boldsymbol{\phi}(S', a)$
    Update visitation count estimates $\hat{N}(S, A)$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[ R + \frac{\beta \sigma_{\max} Z_{\theta/2}}{\hat{N}(S,A)} + \gamma \max_a \mathbf{w}^\top \boldsymbol{\phi}(S', a) - \mathbf{w}^\top \boldsymbol{\phi}(S, A) \right] \boldsymbol{\phi}(S, A)$
    $S \leftarrow S'$, $A \leftarrow A'$
    Exit if *training budget exhausted*
  **end**
  Update visitation count estimates $\hat{N}(S, A)$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[ R + \frac{\beta \sigma_{\max} Z_{\theta/2}}{\hat{N}(S,A)} + \gamma \max_a \mathbf{w}^\top \boldsymbol{\phi}(S', a) - \mathbf{w}^\top \boldsymbol{\phi}(S, A) \right] \boldsymbol{\phi}(S, A)$
**end**

**Algorithm 3:** UpdateWeightsL-UCLS

$\delta \leftarrow R + \gamma \mathbf{w}^\top \boldsymbol{\phi}(S', A') - \mathbf{w}^\top \boldsymbol{\phi}(S, A)$
$\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \boldsymbol{\phi}(S, A)$
$\delta_{\text{var}} \leftarrow \delta + \gamma \mathbf{w}_{\text{var}}{}^\top \boldsymbol{\phi}(S', A') - \mathbf{w}_{\text{var}}{}^\top \boldsymbol{\phi}(S, A)$
$\mathbf{w}_{\text{var}} \leftarrow \mathbf{w}_{\text{var}} + \alpha_{\text{var}} \delta_{\text{var}} \boldsymbol{\phi}(S, A)$
temp $\leftarrow v_{\text{init}}$
$v_{\text{init}} \leftarrow \max(v_{\text{init}}, \mathbf{w}_{\text{var}}{}_1^2, \ldots, \mathbf{w}_{\text{var}}{}_d^2)$
**if** *temp* $\neq v_{init}$ **then**
  $\mathbf{w}_{\text{varInit}} \leftarrow \mathbf{w}_{\text{varInit}} + (v_{\text{init}} - \text{temp})\mathbf{c}$
**end**
**for** *i such that* $\boldsymbol{\phi}(S, A)_i \neq 0$ **do**
  $\mathbf{c}_i \leftarrow (1 - \beta)\mathbf{c}_i$
  $\mathbf{w}_{\text{varInit}} \leftarrow (1 - \beta)\mathbf{w}_{\text{varInit}}$
**end**

**Algorithm 4:** L-UCLS (OFU) for estimating $Q \approx q^*$

**Input:** arbitrary value function weights $\mathbf{w} \in \mathbb{R}^d$
**Input:** discount factor $\gamma \in [0, 1]$
**Parameters:** value function learning rate $\alpha > 0$, variance function learning rate $\alpha_{\text{var}} > 0$, confidence interval width $1 - \theta$, feature map $\boldsymbol{\phi} \colon S \times A \to \mathbb{R}^d$
**Output:** action-value function $Q(S, A) = \mathbf{w}^\top \boldsymbol{\phi}(S, A)$

Initialize $v_{\text{init}} \leftarrow \mathbf{1}$, $c \leftarrow \mathbf{1}$, $\beta \leftarrow 0.001$
Initialize $\mathbf{w}_{\text{var}} \leftarrow \mathbf{0}$, $\mathbf{w}_{\text{varInit}} \leftarrow \mathbf{1}$
**until** *training budget exhausted* **do**
  $S \leftarrow$ initial state of the episode
  **foreach** *action* $a$ **do**
    $u_a \leftarrow \sqrt{\left(1 - \frac{1}{\theta}\right) \left( (\mathbf{w}^\top \boldsymbol{\phi}(S, a))^2 \, \|\boldsymbol{\phi}(S, a)\|^2_{\mathbf{w}_{\text{varInit}}} \right)}$
  **end**
  $A \leftarrow \operatorname{argmax}_a \mathbf{w}^\top \boldsymbol{\phi}(S, a) + u_a$
  **until** *episode terminates* **do**
    Take action $A$, observe $R$, $S'$
    **foreach** *action* $a$ **do**
      $u_a \leftarrow \sqrt{\left(1 - \frac{1}{\theta}\right) \left( (\mathbf{w}^\top \boldsymbol{\phi}(S', a))^2 \, \|\boldsymbol{\phi}(S', a)\|^2_{\mathbf{w}_{\text{varInit}}} \right)}$
    **end**
    $A' \leftarrow \operatorname{argmax}_a \mathbf{w}^\top \boldsymbol{\phi}(S', a) + u_a$
    Run UpdateWeightsL-UCLS
    $S \leftarrow S'$, $A \leftarrow A'$
    Exit if *training budget exhausted*
  **end**
  Run UpdateWeightsL-UCLS
**end**

to time $t$, followed by an additional $\phi$. Then, we can write the $\phi$-pseudocount as

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

These counts can be used to count state visitation or state-action visitation in all OFU methods. We evaluate each variant, which we call S-counts and SA-counts, in our experiments. Pseudocode for SA-counts is presented in Algorithm 1. The counts, $n$, are used to form a reward bonus of the form $r^+ = \frac{\beta}{\sqrt{n}}$, where $\beta$ is a hyperparameter. We sweep over $\beta \in \{0.005, 0.01, 0.05, 0.1, 0.5\}$.

IEQL+ tries to maintain approximate confidence bounds around the optimal value function. It uses a global standard deviation parameter $\sigma_{\max}$ and a confidence interval size parameter $\theta$ to create a reward bonus $r^+ = \frac{\sigma_{\max} Z_{\theta/2}}{\sqrt{n}}$, where $Z_{\theta/2}$ is the value at which the cumulative distribution function of the standard normal distribution has value $1 - \frac{\theta}{2}$. Pseudocode is included in Algorithm 2 We sweep $\sigma_{\max} \in \{0.001, 0.005, 0.01, 0.015\}$, and $\theta \in \{0.5, 0.75, 0.9, 0.95, 0.99\}$.

Our final OFU method is L-UCLS, which also maintains upper confidence bounds on the action-value function. The upper bounds are calculated using a variance estimate which is learned using a secondary set of weights. The secondary weights are updated with a separate learning learning parameter $\alpha_{\text{var}}$, and are scaled by an additional parameter $p = \sqrt{1 - \frac{1}{\theta}}$. Pseudocode is shown in Algorithm 4. We sweep $\alpha_{\text{var}} \in \{0.005, 0.01, 0.05, 0.1, 0.5\}$, and $p \in \{0.05, 0.5, 1, 5, 10\}$.

L-UCLS is the only method we test that learns its value function using Sarsa rather than Q-learning. Since Sarsa does not directly estimate the optimal value function, it may be at a disadvantage in the pure exploration setting that we study here.

### 5.2.2 Thompson sampling

Our first TS method is also L-UCLS. Instead of using the variance estimate to create an upper bound, L-UCLS (TS) uses it to constructs a normal distribution around the mean value estimates. Actions are taken by sampling from

**Algorithm 5:** L-UCLS (TS) for estimating $Q \approx q^*$

**Input:** arbitrary value function weights $\mathbf{w} \in \mathbb{R}^d$
**Input:** discount factor $\gamma \in [0, 1]$
**Parameters:** value function learning rate $\alpha > 0$, variance function learning rate $\alpha_{\text{var}} > 0$, confidence interval width $1 - \theta$, feature map $\boldsymbol{\phi} \colon S \times A \to \mathbb{R}^d$
**Output:** action-value function $Q(S, A) = \mathbf{w}^\top \boldsymbol{\phi}(S, A)$

Initialize $v_{\text{init}} \leftarrow \mathbf{1}$, $c \leftarrow \mathbf{1}$, $\beta \leftarrow 0.001$
Initialize $\mathbf{w}_{\text{var}} \leftarrow \mathbf{0}$, $\mathbf{w}_{\text{varInit}} \leftarrow \mathbf{1}$
**until** *training budget exhausted* **do**
  $S \leftarrow$ initial state of the episode
  **foreach** *action* $a$ **do**
    $u_a \leftarrow \sqrt{\left(1 - \frac{1}{\theta}\right)\left((\mathbf{w}^\top \boldsymbol{\phi}(S, a))^2 \, \|\boldsymbol{\phi}(S, a)\|^2_{\mathbf{w}_{\text{varInit}}}\right)}$
    $v_a \sim \mathcal{N}(\mathbf{w}^\top \boldsymbol{\phi}(S, A), u_a^2)$
  **end**
  $A \leftarrow \text{argmax}_a v_a$
  **until** *episode terminates* **do**
    Take action $A$, observe $R$, $S'$
    **foreach** *action* $a$ **do**
      $u_a \leftarrow \sqrt{\left(1 - \frac{1}{\theta}\right)\left((\mathbf{w}^\top \boldsymbol{\phi}(S', a))^2 \, \|\boldsymbol{\phi}(S', a)\|^2_{\mathbf{w}_{\text{varInit}}}\right)}$
      $v_a \sim \mathcal{N}(\mathbf{w}^\top \boldsymbol{\phi}(S, A), u_a^2)$
    **end**
    $A' \leftarrow \text{argmax}_a v_a$
    Run UpdateWeightsL-UCLS
    $S \leftarrow S'$, $A \leftarrow A'$
    Exit if *training budget exhausted*
  **end**
  Run UpdateWeightsL-UCLS
**end**

**Algorithm 6:** BootstrapQ for estimating $Q \approx q^*$

**Input:** arbitrary value function weights $\mathbf{w_i} \in \mathbb{R}^d$ for $i = 1, 2, \ldots, k$
**Input:** discount factor $\gamma \in [0, 1]$
**Parameters:** learning rate $\alpha > 0$, feature map $\boldsymbol{\phi} : S \times A \to \mathbb{R}^d$
**Output:** k action-value functions $Q_i(S, A) = \mathbf{w_i}^\top \boldsymbol{\phi}(S, A)$

**until** *training budget exhausted* **do**
  $S \leftarrow$ initial state of the episode
  $i \leftarrow$ uniformly select one of the heads **until** *episode terminates* **do**
    Take action $A$, observe $R$, $S'$
    $A' \leftarrow \text{argmax}_a \mathbf{w_i}^\top \boldsymbol{\phi}(S, a)$
    **foreach** $i = 1, 2, \ldots, k$ **do**
      Sample bootstrap mask $m \sim \text{Poisson}(1)$
      $\mathbf{w_i} \leftarrow \mathbf{w_i} + \alpha m[R + \gamma \max_a \mathbf{w_i}^\top \boldsymbol{\phi}(S', a) - \mathbf{w_i}^\top \boldsymbol{\phi}(S, A)]\boldsymbol{\phi}(S, A)$
    **end**
    $S \leftarrow S'$, $A \leftarrow A'$
    Exit if *training budget exhausted*
  **end**
  **foreach** $i = 1, 2, \ldots, k$ **do**
    Sample bootstrap mask $m \sim \text{Poisson}(1)$
    $\mathbf{w_i} \leftarrow \mathbf{w_i} + \alpha m[R + \gamma \max_a \mathbf{w_i}^\top \boldsymbol{\phi}(S', a) - \mathbf{w_i}^\top \boldsymbol{\phi}(S, A)]\boldsymbol{\phi}(S, A)$
  **end**
**end**

the distribution of action-values in the current state and selecting the action with the highest sampled value. The pseudocode, Algorithm 5, is the same as that of L-UCLS (OFU) above in, but uses $\boldsymbol{u}_a$ to form a Gaussian distribution over action-values instead of simply adding it to the mean value estimates.

We also test an implementation of BootstrapDQN that uses Q-learning. BootstrapQ learns an ensemble of $k = 10$ action-value functions. Actions are selected greedily with respect to one of the value functions, which is randomly selected at the beginning of the episode. The 10 action-value functions are updated every time-step, but their updates are multiplied by a Poisson(1) random variable which acts to maintain the statistical validity of interpreting the ensemble as a bootstrap. Pseudocode is shown in Algorithm 6.

### 5.2.3 Learning progress methods

**Algorithm 7:** Optimistic Initialization for estimating $Q \approx q^*$

**Input:** arbitrary value function weights $\mathbf{w} \in \mathbb{R}^d$
**Input:** discount factor $\gamma \in [0, 1]$
**Parameters:** learning rate $\alpha > 0$, feature map $\boldsymbol{\phi} : S \times A \to \mathbb{R}^d$, value bonus $\mathbf{w}_{\text{bonus}} \in \mathbb{R}^d_{>0}$
**Output:** action-value function $Q(S, A) = \mathbf{w}^\top \boldsymbol{\phi}(S, A)$

$\mathbf{w} \leftarrow \mathbf{w} + \mathbf{w}_{\text{bonus}}$
**until** *training budget exhausted* **do**
  $S \leftarrow$ initial state of the episode
  $A \leftarrow \text{argmax}_a \mathbf{w}^\top \boldsymbol{\phi}(S, a)$
  **until** *episode terminates* **do**
    Take action $A$, observe $R$, $S'$
    $A' \leftarrow \text{argmax}_a \mathbf{w}^\top \boldsymbol{\phi}(S', a)$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma \max_a \mathbf{w}^\top \boldsymbol{\phi}(S', a) - \mathbf{w}^\top \boldsymbol{\phi}(S, A)]\boldsymbol{\phi}(S, A)$
    $S \leftarrow S'$, $A \leftarrow A'$
    Exit if *training budget exhausted*
  **end**
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha[R - \mathbf{w}^\top \boldsymbol{\phi}(S, A)]\boldsymbol{\phi}(S, A)$
**end**

We test two learning progress methods, Random Network Distillation (RND) and Optimistic Initialization (OptInit). Pseudocode for RND is shown in Algorithm 8. RND's auxiliary function is from the domain of the state-

**Algorithm 8:** Random Network Distillation for estimating $Q \approx q^*$

**Input:** arbitrary value function weights $\mathbf{w} \in \mathbb{R}^d$
**Input:** discount factor $\gamma \in [0, 1]$
**Input:** arbitrary secondary function weights $\mathbf{U} \in \mathbb{R}^{d \times k}$
**Parameters:** learning rate $\alpha > 0$, reward scaling parameter $\beta > 0$,
feature map $\boldsymbol{\phi} : S \times A \to \mathbb{R}^d$
**Output:** action-value function $Q(S, A) = \mathbf{w}^\top \boldsymbol{\phi}(S, A)$

Randomly initialize $\boldsymbol{U}^* \in \mathbb{R}^{d \times k}$
**until** *training budget exhausted* **do**
  $S \leftarrow$ initial state of the episode
  $A \leftarrow \text{argmax}_a \mathbf{w}^\top \boldsymbol{\phi}(S, a)$
  **until** *episode terminates* **do**
    Take action $A$, observe $R$, $S'$
    $A' \leftarrow \text{argmax}_a \mathbf{w}^\top \boldsymbol{\phi}(S', a)$
    ▷ `update secondary function`
    $\boldsymbol{\delta}_U \leftarrow \mathbf{U}\boldsymbol{\phi}(S, A) - \boldsymbol{U}^*\boldsymbol{\phi}(S, A)$
    $\boldsymbol{U} \leftarrow \boldsymbol{U} + \alpha \boldsymbol{\delta}_U$
    ▷ `update value function`
    $\sigma \leftarrow \mathit{StandardDeviation}(\boldsymbol{\delta}_{U,1:t})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[ R + \frac{\beta \|\boldsymbol{\delta}_U\|_2}{\sigma} + \gamma \max_a \mathbf{w}^\top \boldsymbol{\phi}(S', a) - \mathbf{w}^\top \boldsymbol{\phi}(S, A) \right] \boldsymbol{\phi}(S, A)$
    $S \leftarrow S'$, $A \leftarrow A'$
    Exit if *training budget exhausted*
  **end**
  ▷ `update secondary function`
  $\boldsymbol{\delta}_U \leftarrow \mathbf{U}\boldsymbol{\phi}(S, A) - \boldsymbol{U}^*\boldsymbol{\phi}(S, A)$
  $\boldsymbol{U} \leftarrow \boldsymbol{U} + \alpha \boldsymbol{\delta}_U$
  ▷ `update value function`
  $\sigma \leftarrow \mathit{StandardDeviation}(\boldsymbol{\delta}_{U,1:t})$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[ R + \frac{\beta \|\boldsymbol{\delta}_U\|_2}{\sigma} - \mathbf{w}^\top \boldsymbol{\phi}(S, A) \right] \boldsymbol{\phi}(S, A)$
**end**

value function, for RND-S, or action-value function, for RND-SA, to a $k = 10$-dimensional vector. The target function is a (#heads $\times$ #features) matrix with uniformly random values between 0 and 1, which is generated at the beginning of each trial. The auxiliary function is learned using stochastic gradient descent with the same learning step-size as Q-learning. The reward bonus is based on the L2-norm of the error between the auxiliary function's value and the target function's value. We normalize the reward bonus by dividing it by the standard deviation of the norm of the error, then scale it by a hyperparameter $\beta \in \{0.005, 0.01, 0.05, 0.1, 0.5\}$.

Our implementation of OptInit is shown in Algorithm 7. It is equivalent to Q-learning, but with weights initialized to a certain value rather than zero. We sweep over initial weights $w \in \{-1, 1, 10, 20, 50, 100\}$.

### 5.2.4 Noisy methods

**Algorithm 9:** Epsilon-greedy Q-learning for estimating $Q \approx q^*$

**Input:** arbitrary value function weights $\mathbf{w} \in \mathbb{R}^d$
**Input:** discount factor $\gamma \in [0, 1]$
**Parameters:** learning rate $\alpha > 0$, feature map $\boldsymbol{\phi} : S \times A \to \mathbb{R}^d$, small $\epsilon > 0$
**Output:** action-value function $(S, A) = \mathbf{w}^\top \boldsymbol{\phi}(S, A)$

**until** *training budget exhausted* **do**
  $S \leftarrow$ initial state of the episode
  Select action $A$ according to epsilon-greedy
  **until** *episode terminates* **do**
    Take action $A$, observe $R$, $S'$
    Select action $A'$ according to epsilon-greedy
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma \max_a \mathbf{w}^\top \boldsymbol{\phi}(S', a) - \mathbf{w}^\top \boldsymbol{\phi}(S, A)]\boldsymbol{\phi}(S, A)$
    $S \leftarrow S'$, $A \leftarrow A'$
    Exit if *training budget exhausted*
  **end**
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha[R - \mathbf{w}^\top \boldsymbol{\phi}(S, A)]\boldsymbol{\phi}(S, A)$
**end**

We test Epsilon-greedy and NoisyNets. For Epsilon-greedy, we sweep over $\epsilon \in \{0.015625, 0.03125, 0.0625, 0.125, 0.25, 0.5\}$. Powers of two were cho-

**Algorithm 10:** NoisyNets for estimating $Q \approx q^*$

**Input:** arbitrary value function mean weights $\mathbf{w} \in \mathbb{R}^d$
**Input:** arbitrary value function noise weights $\mathbf{u} \in \mathbb{R}^d$
**Input:** discount factor $\gamma \in [0, 1]$
**Parameters:** learning rate $\alpha > 0$, feature map $\boldsymbol{\phi} \colon S \times A \to \mathbb{R}^d$
**Output:** action-value function $Q(S, A) = (\mathbf{w} + (\mathbf{u} \odot \boldsymbol{\xi}))^\top \boldsymbol{\phi}(S, A)$

**until** *training budget exhausted* **do**
  $S \leftarrow$ initial state of the episode
  $\boldsymbol{\xi} \leftarrow d$ independent standard normal RVs
  $A \leftarrow \operatorname{argmax}_a (\mathbf{w} + (\mathbf{u} \odot \boldsymbol{\xi}))^\top \boldsymbol{\phi}(S, a)$
  **until** *episode terminates* **do**
    Take action $A$, observe $R$, $S'$
    $\boldsymbol{\xi}' \leftarrow d$ independent standard normal RVs
    $A' \leftarrow \operatorname{argmax}_a (\mathbf{w} + (\mathbf{u} \odot \boldsymbol{\xi}'))^\top \boldsymbol{\phi}(S', a)$
    $\boldsymbol{\xi}^{(2)} \leftarrow d$ independent standard normal RVs
    $\boldsymbol{\xi}^{(3)} \leftarrow d$ independent standard normal RVs
    $\delta \leftarrow R + \gamma \max_a \left((\mathbf{w} + (\mathbf{u} \odot \boldsymbol{\xi}^{(2)}))^\top \boldsymbol{\phi}(S', a)\right) - \left(\left(\mathbf{w} + \left(\mathbf{u} \odot \boldsymbol{\xi}^{(3)}\right)\right)^\top \boldsymbol{\phi}(S, A)\right)$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \boldsymbol{\phi}(S, A)$
    $\boldsymbol{\xi}^{(4)} \leftarrow d$ independent standard normal RVs
    $\mathbf{u} \leftarrow \mathbf{u} + \alpha \delta \left(\boldsymbol{\phi}(S, A) \odot \boldsymbol{\xi}^{(4)}\right)$
    $S \leftarrow S'$, $A \leftarrow A'$
    Exit if *training budget exhausted*
  **end**
  $\boldsymbol{\xi}^{(5)} \leftarrow d$ independent standard normal RVs
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[R - \left(\mathbf{w} + \left(\mathbf{u} \odot \boldsymbol{\xi}^{(5)}\right)\right)^\top \boldsymbol{\phi}(S, A)\right] \boldsymbol{\phi}(S, A)$
  $\boldsymbol{\xi}^{(6)} \leftarrow d$ independent standard normal RVs
  $\mathbf{u} \leftarrow \mathbf{u} + \alpha \left[R - \left(\mathbf{w} + \left(\mathbf{u} \odot \boldsymbol{\xi}^{(6)}\right)\right)^\top \boldsymbol{\phi}(S, A)\right] \boldsymbol{\phi}(S, A)$
**end**

sen to sweep over a similar number of values as the other methods. Pseudocode for Epsilon-greedy can be found in Algorithm 9.

NoisyNets learns a value function with parameters for the mean estimate as well as a noise variable for each action and feature. Action-values are calculated according to $Q(s, a) = w + (u \odot \xi))^\top \phi(S, A)$, where $\xi$ is drawn from a standard normal distribution, including when updating the value function and when selecting actions. Here $\odot$ is used to denote element-wise multiplication. NoisyNets does not use any additional hyperparameters. We present pseudocode for NoisyNets in Algorithm 10.

### 5.2.5 Baselines

**Algorithm 11:** Softmax Actor-Critic for estimating $\pi \approx \pi^*$

**Input:** arbitrary value function weights $\mathbf{w} \in \mathbb{R}^m$
**Input:** arbitrary policy weights $\mathbf{u} \in \mathbb{R}^n$
**Input:** discount factor $\gamma \in [0, 1]$
**Parameters:** value function learning rate $\alpha > 0$, policy learning rate $\beta > 0$, value function feature map $\boldsymbol{\phi}: S \to \mathbb{R}^m$, policy feature map $\boldsymbol{\psi}: S \times A \to \mathbb{R}^n$
**Output:** policy $\pi(S, A) = \text{softmax}(\boldsymbol{u}^\top \boldsymbol{\psi}(S, A)) = e^{\boldsymbol{u}^\top \boldsymbol{\psi}(S,A)} / \sum_a e^{\boldsymbol{u}^\top \boldsymbol{\psi}(S,a)}$

**until** *training budget exhausted* **do**
  $S \leftarrow$ initial state of the episode
  $A \sim \text{softmax}(\boldsymbol{u}^\top \boldsymbol{\psi}(S, \cdot))$
  **until** *episode terminates* **do**
    Take action $A$, observe $R$, $S'$
    $A' \sim \text{softmax}(\boldsymbol{u}^\top \boldsymbol{\psi}(S', \cdot))$
    $\delta \leftarrow R + \gamma \max_a \mathbf{w}^\top \boldsymbol{\phi}(S', a) - \mathbf{w}^\top \boldsymbol{\phi}(S, A)$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \boldsymbol{\phi}(S, A)$
    $\mathbf{u} \leftarrow \mathbf{u} + \beta \delta \nabla_u \ln \text{softmax}(\boldsymbol{u}^\top \boldsymbol{\psi}(S, A))$
    $S \leftarrow S'$, $A \leftarrow A'$
    Exit if *training budget exhausted*
  **end**
  $\delta \leftarrow R - \mathbf{w}^\top \boldsymbol{\phi}(S, A)$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \boldsymbol{\phi}(S, A)$
  $\mathbf{u} \leftarrow \mathbf{u} + \beta \delta \nabla_u \ln \text{softmax}(\boldsymbol{u}^\top \boldsymbol{\psi}(S, A))$
**end**

We include two baselines, a random agent and a Softmax Actor-critic

(SoftmaxAC) agent. The random agent always takes random actions.

The SoftmaxAC (Sutton and Barto 2018) agent learns the optimal policy by stochastic gradient descent. Unlike other policy-gradient algorithms, actor-critic algorithms use a value function estimate, called the critic, to reduce the variance of their gradient estimates. SoftmaxAC's policy is parameterized by a softmax function over learned preference values. The preference values and the critic's state-values are both learned using the same features, but we sweep over their learning rates separately, using the same values as the learning rates for other agents. Pseudocode is shown in Algorithm 11.

## 5.3 Summary

In this chapter we described our empirical evaluation scheme, which includes six environments, twelve exploration methods, and two baselines. The exploration methods modify a base Q-learning agent that uses linear function approximation to represent its action-value estimates. We sweep over three different tile-coding schemes for each environment depending on the dimensionality of the state-space. Each tile-coding scheme generates roughly the same number of features as other schemes for the same environment. The environments are based on the difficult exploration properties we proposed in Chapter 3, while the agents are selected from the categories we presented in Chapter 4.

# Chapter 6

# Results

In this chapter we discuss the results from the experiments in the previous chapter. These experiments are meant to provide a sanity check for exploration methods. If a method cannot avoid the traps laid in these toy environments, it is unlikely that they will be able to avoid the same traps in much more complicated environments. So success in these environments is a requirement for any general exploration method, while failure in any one is a cause for concern. We roughly swept each method's hyperparameters to get a general sense for their potential performance. We did not finely tune the parameters because that requires much computation and many samples from the environment. The costs associated with such fine-tuning often overcome the sample-efficiency from exploration.

The results are organized into a main analysis, where we present the performance of each exploration method on each of the environments. We then provide a supplementary analysis where we discuss the robustness of each method to their hyperparameter choices and present a statistical analysis of the overall results. We close with a discussion of the most important observations that result from these experiments and their implications for the future of exploration research in RL.

## 6.1 Main analysis

We present the results of our experiments on each environment in this section. We begin by highlighting some interesting overall patterns, then cover each environment's results in more detail.

### 6.1.1 Highlights

Overall, L-UCLS (OFU) performed very well in all environments except VarianceWorld, where its performance was mixed. This refutes our earlier hypothesis that learning with Sarsa might disadvantage L-UCLS in the pure exploration setting. Two others, L-UCLS (TS) and SoftmaxAC, learned near-optimal policies in all the environments except AlpineSki. The other methods had mixed performance, typically performing well in some environments and poorly in others.

NoisyNets and Epsilon-greedy exploration add noise to agent's behaviour quite differently. As a result, their learned behaviour is not as similar as the methods in other groups. Similarly, OptInit differs from the other learning progress methods in that it only changes the value estimates at one point in time. Other learning progress methods augment the reward with an exploration bonus that changes over time. The following results suggest that exploratory behaviour is usually more strongly influenced by the presence of a decaying positive reward bonus than by the decay rate of the reward bonus.

Among OFU methods, IEQL+ and SA-counts had almost the same distributions over test performance. There is only one major difference between these methods, which is that IEQL+ initializes its value function estimate so that the values represent an approximate upper confidence bound from the first time-step. The similar learned behaviour of IEQL+ and SA-counts suggests that this initial optimism does not impact learning performance when used in conjunction with an optimistic reward bonus.

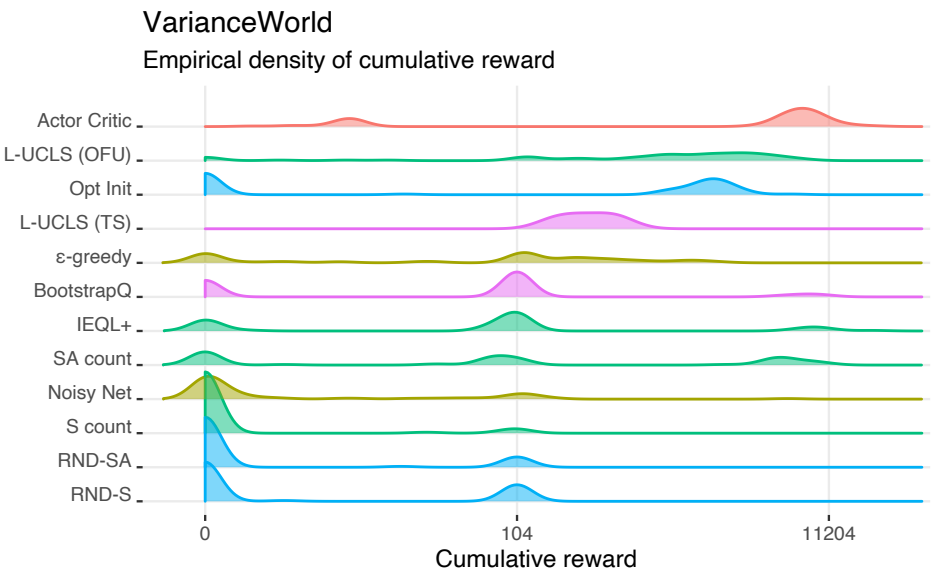


Figure 6.1: This plot shows a distribution over the reward accumulated by each agent over the 100000 time-step test phase. Agents are colored according to their underlying heuristics, and sorted by their median performance. The two non-zero vertical lines correspond to the rewards accumulated by the policy that always goes left and by the optimal policy that always goes right, respectively. Only runs from the best hyperparameter settings are shown. We calculate the plotted densities using the `ggridges` package (Wilke 2018) for `R` (R Core Team 2019), and data from 72 independent runs.

### 6.1.2 High-variance reward: VarianceWorld

VarianceWorld is a short corridor with a large, high-variance reward at the right end, and a small, low variance reward on the left. Agents must be able to sample the high-variance reward sufficiently many times to correctly estimate the return from high-value states. None of the tested methods were able to consistently find a near-optimal policy. In fact, for a substantial proportion of runs, all methods except L-UCLS (TS) and SoftmaxAC learned a policy that got stuck in a region of state-space and failed to terminate a single episode.

We plot the results of our experiments on VarianceWorld in figure 6.1. Only four methods had median performance better than random: SoftmaxAC, OptInit, and both L-UCLS variants. Epsilon-greedy learned policies with highly varying performance, which could be because the value functions are not necessarily consistent with the values of the greedy policy. Such value functions could get stuck temporarily in a region of state space, but might be able to escape due to the stochastic effect of actions.

The performance of the OFU methods is quite spread out. IEQL+, and SA-counts all have tri-modal distributions with peaks at 0, 104, and 11204, while S-counts is bimodal around the lower two peaks. L-UCLS (OFU) has a peak around 0, but the majority of its cumulative rewards are between 104 and 11204. The RND methods got stuck for 0 reward in most of their runs, but were able reach the lower reward in other runs. OptInit also had mixed performance, with peaks at 0 and between 104 and 11204.

The results in VarianceWorld are more mixed than in any of the other environments. It is possible that by choosing a different distribution for the high-variance reward, the agents could have learned more consistently separated policies.

### 6.1.3 Misleading reward: Antishaping

Figure 6.2 shows the results of our experiments in the Antishaping environment. Antishaping is a longer corridor with a small reward on the non-

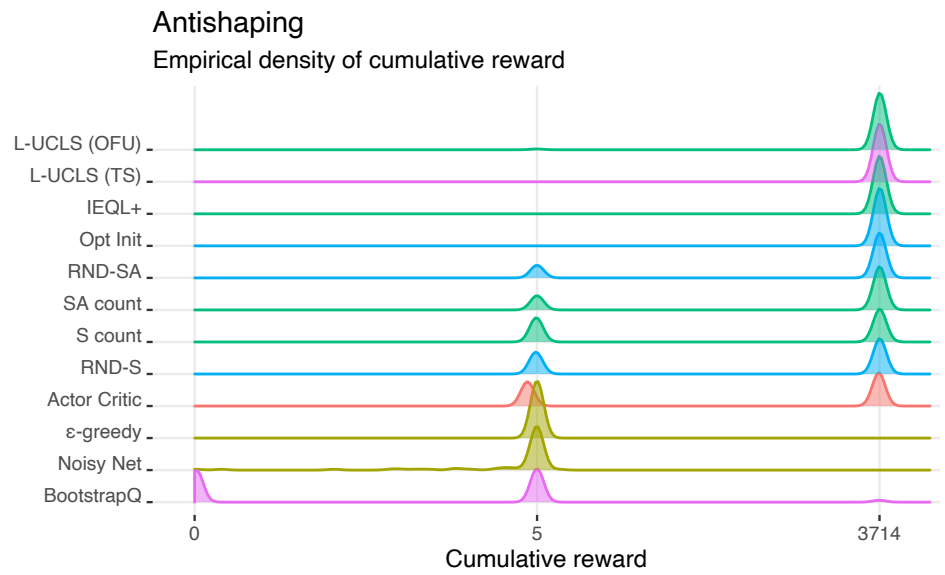


Figure 6.2: This plot shows a distribution over the reward accumulated by each agent over the 100000 time-step test phase. Agents are colored according to their underlying heuristics, and sorted by their median performance. The two non-zero vertical lines correspond to the rewards accumulated by the policy that stays at the left edge and by the optimal policy that always goes right, respectively. Only runs from the best hyperparameter settings are shown. We calculate the plotted densities using the `ggridges` package for `R`, and data from 72 independent runs.

terminating leftmost state that decreases towards the right end of the corridor. Agents must be able to ignore the decreasing behaviour in the reward and travel all the way to the right-end of the corridor where the episode will terminate with a large reward. Both L-UCLS methods, OptInit, and IEQL+ consistently solved this environment. IEQL+ and SA-counts have different performance in this environment because the exploratory effect of optimistic initialization dominates the misleading effect of the small environmental rewards. As a result, SA-counts performs similarly to the most of the other OFU and Learning Progress reward-bonus methods, which usually learn a near-optimal policy, but sometimes learn to stay at the left-side of the environment.

The rest of the methods had lower median performance than the random agent. The noise-based exploration methods both consistently learn to stay on the left side of the environment. Interestingly, BootstrapQ appears to learn policies that either stay near the left or get stuck somewhere in the middle of the environment.

Our experiments in the Antishaping environment suggest that Noisy methods may not sufficiently explore when there is a misleading reward signal.

### 6.1.4 Sparse reward: Sparse MountainCar

Figure 6.3 shows the results of our experiments in the Sparse MountainCar environment. In Sparse MountainCar, an agent must learn to follow a spiral path radiating outwards from the start state to the goal. The only environmental reward signal is found when the agent reaches the goal state and terminates the episode. Epsilon-greedy, OptInit, both L-UCLS variants, BootstrapQ, and SoftmaxAC all learned strong policies in this domain, sometimes even learning a policy that is better than our baseline near-optimal policy. The other methods learn policies that appear to reach the goal similarly frequently to the random agent.

These results support our earlier observation that the presence of a decaying positive reward bonus is more important than the rate at which the bonus

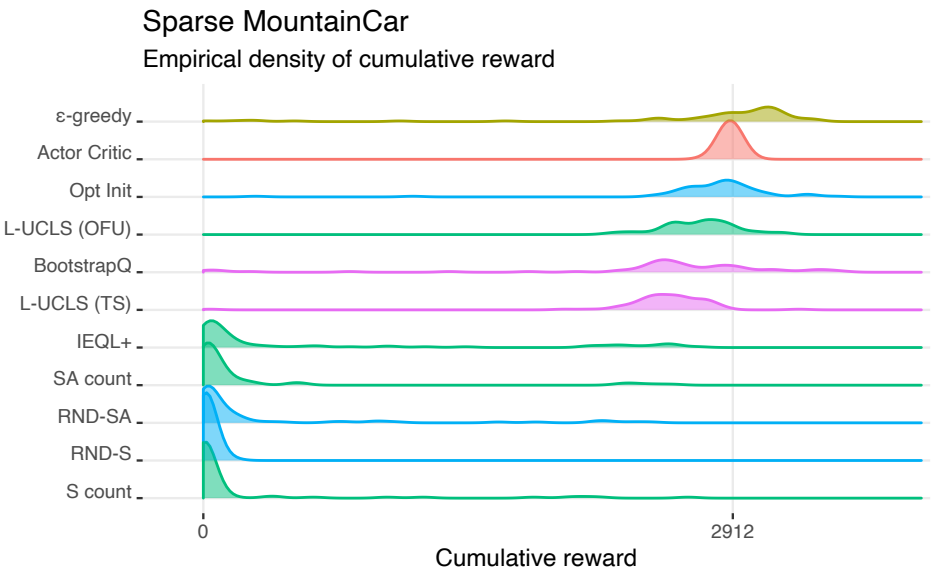


Figure 6.3: This plot shows a distribution over the reward accumulated by each agent over the 100000 time-step test phase. Agents are colored according to their underlying heuristics, and sorted by their median performance. The vertical line at 2912 corresponds to the reward accumulated by a near-optimal policy. Only runs from the best hyperparameter settings are shown. We calculate the plotted densities using the `ggridges` package for `R`, and data from 72 independent runs.

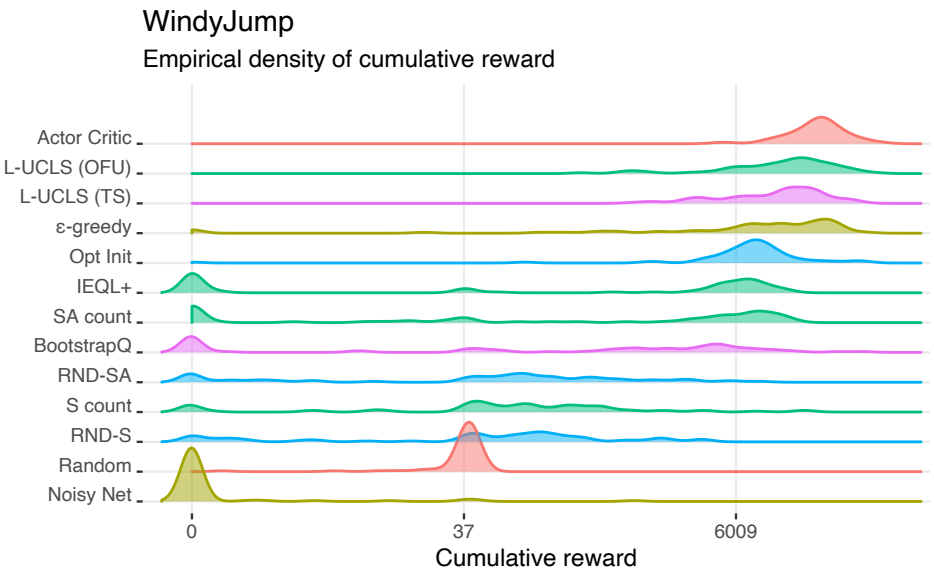


Figure 6.4: This plot shows a distribution over the reward accumulated by each agent over the 100000 time-step test phase. Agents are colored according to their underlying heuristics, and sorted by their median performance. The two non-zero vertical lines correspond to the rewards accumulated by the policy that always goes left and by the near-optimal policy that always goes right, respectively. For ease of visualization, the few negative accumulated values are clipped to −0.01. Only runs from the best hyperparameter settings are shown. We calculate the plotted densities using the `ggridges` package for `R`, and data from 72 independent runs.

decays. In contrast to the results in the Antishaping environment, IEQL+ performs similarly to SA-counts despite the strong performance of OptInit, suggesting that reward bonuses may interact poorly with Q-learning in sparse reward environments.

### 6.1.5 High-variance transitions: WindyJump

Figure 6.4 shows the results of our experiments in the Antishaping environment. We note that clipping the values does not affect the visualization except by hiding long tails created by a small handful of points.

Success in WindyJump depends on the agent learning to explore an area of the environment that has highly varying transitions. All methods other than NoisyNets were able to identify the positive reward on the right side of the environment, achieving near-optimal reward in some runs and outperform-

ing the random agent. Surprisingly, the top-performing methods SoftmaxAC, Epsilon-greedy, OptInit, and both L-UCLS variants frequently performed better than the reference policy that always moves to the right, suggesting that they often learn a more sophisticated policy than the reference. The rest of the methods are separated by how often they got stuck and received 0 reward, and how often they learned to go towards the large reward. Unlike in most of the other environments, very few of the exploration methods learn policies that go towards the small reward. Instead, possibly due to the stochasticity in the environment, BootstrapQ, S-counts, and both RND methods learned policies whose is spread out between the two reference policies.

In contrast with the high-variance reward domain VarianceWorld, all exploration methods learned a near-optimal policy in at least a few runs. We interpret these results as evidence that high-variance rewards poses a much more difficult exploration problem than high-variance transitions in domains with small state-action spaces.

### 6.1.6 Antagonistic transitions: AlpineSki

Figure 6.5 shows the results of our experiments in the AlpineSki environment. The difficult part of AlpineSki is that the agent must traverse for many timesteps before it can ski down and receive reward. Agents that learn stochastic behaviour policies, or agents that explore by adding noise in some fashion, are unlikely to be able to solve this problem. Indeed, we find that Epsilon-greedy, SoftmaxAC, and L-UCLS (TS) learn to ski down immediately to receive the small reward.

Despite incorporating randomness into their value estimates, BootstrapQ learned near-optimal policies in some runs, and NoisyNets learned near-optimal policies in almost all of the runs.

As in WindyJump and MountainCar, L-UCLS (OFU) and OptInit consistently learn the optimal policy. The remaining reward bonus methods learn a split between near-optimal policies and the policy that skis down immediately. Of these, IEQL+ has the strongest median performance, which maybe

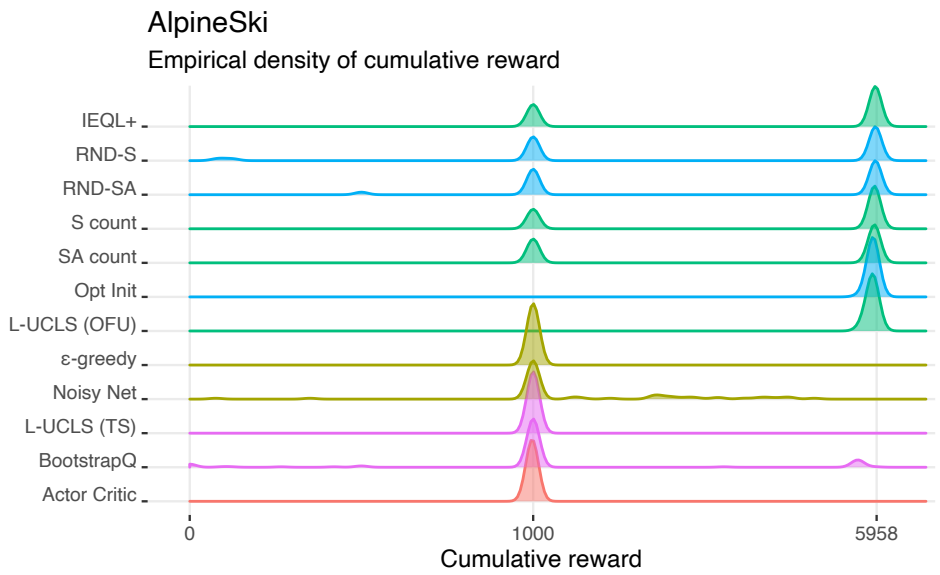


Figure 6.5: This plot shows a distribution over the reward accumulated by each agent over the 100000 time-step test phase. Agents are colored according to their underlying heuristics, and sorted by their median performance. The two non-zero vertical lines correspond to the rewards accumulated by the policy that skis down immediately and by the optimal policy, respectively. Only runs from the best hyperparameter settings are shown. We calculate the plotted densities using the `ggridges` package for `R`, and data from 72 independent runs.

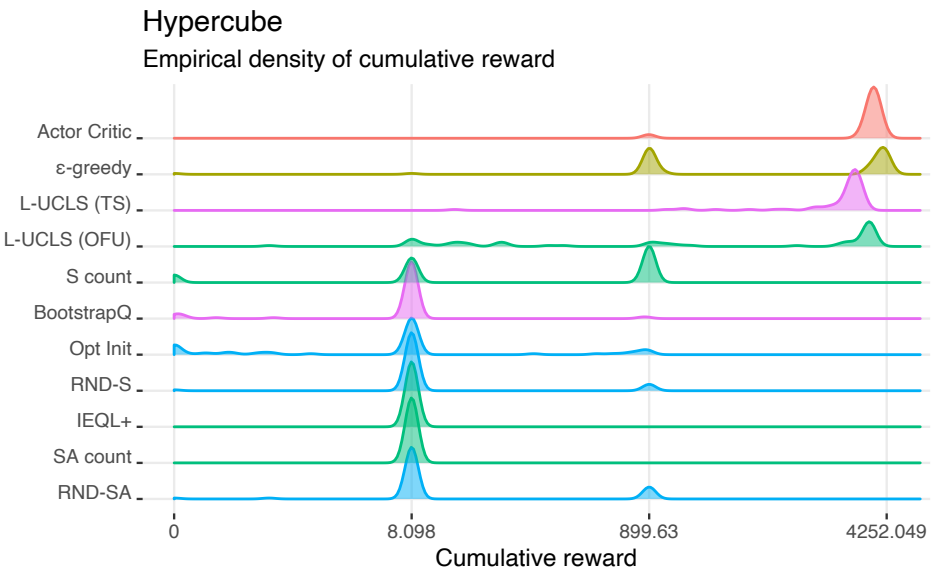


Figure 6.6: This plot shows a distribution over the reward accumulated by each agent over the 100000 time-step test phase. Agents are colored according to their underlying heuristics, and sorted by their median performance. The vertical lines correspond to the rewards accumulated by policies that directly move towards a corner that touches 0, 1, 2, or 3 walls. Only runs from the best hyperparameter settings are shown. We calculate the plotted densities using the `ggridges` package for `R`, and data from 72 independent runs.

attributable to its initially inflated value estimates.

The AlpineSki environment represents the strongest argument against learning policies that are necessarily stochastic. Policy gradient methods, Noisy methods, and Thompson Sampling all fall into this category. However, as NoisyNets demonstrates, in some cases the agent can learn that the stochasticity should be reduced to a point where the policy is effectively deterministic.

### 6.1.7 Large state-action space: Hypercube

Figure 6.6 shows the results of our experiments in the Hypercube environment. Hypercube is relatively large, symmetrical along each dimension, and possesses a shaping reward to help guide the agents along the optimal policy. As in most of the environments, SoftmaxAC and both variants of L-UCLS perform quite well. Somewhat surprisingly, Epsilon-greedy also performs well.

The results here and in Sparse MountainCar suggest that Epsilon-greedy is effective at exploring large state-action spaces that do not have tricky reward or transition dynamics. NoisyNets regularly learns a policy that stays near the intersection of two walls.

The remaining methods, including the random agent, learn policies that tend to stay near a single wall for the duration of the testing phase. Some methods, especially S-counts, are able to consistently reach a corner between two walls, but do not learn to terminate the episode and receive a larger reward. Even OptInit, which typically sweeps over the entire state-action space during learning, was not able to identify the optimal policy in this domain.

When using incremental, model-free methods like Q-learning, the state-action space must be swept several times to propagate the rewards, which are experienced at the end of episodes, through the value function to the region of state space where the agent begins an episode. Since we freeze the exploration mechanisms during the testing phase, we suspect that the non-performant agents are failing to explore sufficiently during the training phase, rather than struggling to balance exploration and exploitation during testing.

## 6.2 Supplementary analyses

Here we present two additional perspectives on the experimental data. First, we discuss the impact of hyperparameter choices on the performance of each exploration method. We then perform a statistical analysis on the data to determine whether any one method consistently outperformed the others.

### 6.2.1 Parameter Study

We present plots for the parameter sweeps in Appendix B. Many parameter settings for most methods learned policies that do not achieve any reward. There are two exceptions. SoftmaxAC achieved strictly non-zero reward on Antishaping, AlpineSki, and Hypercube, which constitute half of the environ-

ments we tested. Epsilon-greedy also achieved non-zero reward in AlpineSki.

The majority of the hyperparameter settings tend to result in agents learning a policy that gets stuck in a region of the state space. This indicates that our swept hyperparameter values successfully covered the space of effective settings.

Unfortunately most of the hyperparameter settings failed to learn a near-optimal policy in any environment. In practical settings, the desire for efficient exploration is balanced by the aversion to sweeping over hyperparameters — if an agent has to be run hundreds of times to identify successful hyperparameters it limits the benefits offered by efficient exploration. This parameter study highlights the need for algorithms that are robust to hyperparameter selection while being able to explore efficiently. In a practical sense, hyperparameter tuning may be just as undesirable as inefficient exploration.

### 6.2.2 Statistical Analysis

Our data is a collection of multiple independent samples from the best hyperparameter settings for each agent-environment pair. In statistical terms, we call the exploration methods *treatments* and the environments *blocks*. Due to the simple environment design, performance is multimodally distributed, even when conditioned on the exploration method and environment. We therefore model our data using quantile regression on the median performance with the following form:

$$y_{ijk} = \alpha_i + \beta_j + (\alpha\beta)_{ij} + \epsilon_{ijk},$$

where $i$, $j$, and $k$ index the exploration methods, environments, and experimental runs respectively. The variable $y$ represents accumulated reward. The right-hand variables $\alpha_i$, $\beta_j$, and $(\alpha\beta)_{ij}$ are coefficients describing the relationship between the median accumulated reward during the testing phase and the exploration method, environment, and interactions respectively. Finally, $\epsilon$ is a noise term. We verify our inclusion of interaction terms numerically

by comparing our chosen model with the same model without interaction terms using a rank-test statistic described in Gutenbrunner et al. (1993) and implemented in the `R` package `quantreg` (Koenker 2019).

We perform a series of T-tests (see Appendix C) to determine the significance of our model fit, using a kernel method implemented in `quantreg` to estimate the standard errors of the coefficients. Almost all of the coefficients are highly significant, with p-values equal to zero (due to rounding) in the output produced by `quantreg`. The few remaining coefficients only have p-values not near zero (ie, not $p \ll 0.05$) because the coefficients themselves are very close to 0. These results confirm that no exploration method dominates all others, with each method under-performing relative to other methods in at least one of the tested environments. The results also suggest that each environment poses distinct exploration challenges.

## 6.3 Discussion

We conducted a series of experiments to study the empirical behaviour of several exploration methods in various distinctly challenging environments. We examined the sensitivity of each method to its hyperparameter settings, and conducted a statistical test to see if any one method outperformed the others. Overall, SoftmaxAC, L-UCLS (OFU), Epsilon-greedy, and OptInit performed well in many environments.

We found that the performance of OFU and Learning Progress methods was typically not divided by the categories laid out in Chapter 4. Instead, methods that used inflated values to select actions, such as L-UCLS (OFU) and OptInit, tended to perform better than methods that used a reward bonus, such as RND. These results suggest that algorithm-level decisions impact empirical performance more strongly than theoretical considerations like the decay rate of exploration.

We can study the combined effect of inflated values and reward bonuses by examining the performance of IEQL+. IEQL+'s typically poor perfor-

mance suggests that reward bonuses are not as effective at inducing exploration as using already-inflated values to explore. This is probably because reward bonuses have to be propagated through the value function to encourage exploration — the agent is only rewarded after the desired exploratory behaviour occurs. In contrast, inflated values anticipate additional reward where the agent ought to explore, causing it to explore automatically simply by acting greedily.

IEQL+ performs similarly to reward bonus methods in VarianceWorld, WindyJump, AlpineSki, and MountainCar, despite being initialized with inflated values. There is no clear split in Hypercube, but in Antishaping IEQL+ instead performs similarly to the inflated value methods. This pattern of results suggests that reward bonuses distract inflated value agents from exploring in most methods, and additionally that the distracting effect is not present when there is a misleading reward signal in the environment. Despite this intriguing observation, on the whole our results suggest avoiding using positive, decaying reward bonuses to encourage exploration.

The Noisy and Thompson Sampling methods all have randomized behaviour, and performed similarly in WindyJump, Mountain Car, and Hypercube. Epsilon-greedy and L-UCLS (TS) performed better than NoisyNets and BootstrapQ respectively. Since the pairs from both categories used similar heuristic strategies, we believe that Epsilon-greedy and L-UCLS (TS) were more successful simply because of their algorithmic implementation.

SoftmaxAC is incredibly successful in all of the environments we tested other than AlpineSki. We note that the broader class of policy gradient methods, including SoftmaxAC, find a near-optimal policy using gradient descent. Their objective functions depend on the environment and function approximation scheme, and are usually non-convex, so policy gradient methods might find an arbitrarily bad policy in general. Identifying the settings in which policy gradient methods find a near-optimal policy is an ongoing area of research, for example see Bhandari et al. (2019).

Despite the occasional success of randomized behaviour methods, espe-

cially SoftmaxAC, they uniformly fail to explore in AlpineSki. AlpineSki requires the agent to take a very specific series of mutually consistent actions to reach a reward. When an agent explores by randomizing its behaviour, it is very unlikely that it will sample the correct sequence of actions.

One method, BootstrapQ, attempts to randomize behaviour while exploring consistently. By behaving greedily according to a plausible value function that it randomly selects at the beginning of each episode, BootstrapQ was able to identify a near-optimal policy in 13.2% of runs in AlpineSki. Despite being designed to explore consistently, our results indicate that a statistical bootstrap is an effective way of incorporating randomness into Q-learning.

We expected the high-variance reward environment VarianceWorld to be relatively straightforward for agents to solve since many of the exploration strategies used by the methods we tested have their roots in algorithms for multi-armed bandits, where the exploration problem is largely based on the variance in rewards. Even though VarianceWorld resembles a two-armed bandit in the sense that there are two sources of reward that are easy to experience at a delay of only 10 time-steps, our tested methods were not able to find consistently good policies. Our results suggest that the difficulty posed by high-variance rewards may be meaningfully different in bandits and MDPs, even when the MDPs in question can be traversed in few time-steps.

## 6.4 Summary

This chapter discussed the results of our empirical study. We compared the performance of several exploration methods on six toy environments that each posed a separate challenge to exploring agents. All the methods were more effective in some environments than in others. Our statistical analysis found that L-UCLS (OFU) learned the highest-performing policies across all six toy environments.

Our conceptual categorization of exploration environments was not mirrored in the experimental results. Rather, methods performed similarly if

they used a reward bonus, chose actions according to inflated values, or randomized their learned policies. This grouping suggests that at this time, algorithmic design choices have a greater effect on agent learning and behaviour than the underlying heuristic used to drive exploration.

In our parameter study, we found that the performance was quite poor for the vast majority of hyperparameter settings and environments. We interpret this poor performance as a reminder that efficient exploration is only useful in practice if the practitioner does not have to manually explore a large hyperparameter space.

To the best of our knowledge this is the first empirical study that uses contrasting environments to investigate the properties of exploration methods and environments that affect the performance of learned policies.

# Chapter 7

# Conclusion & Future Work

In this thesis we studied the relationship between exploration heuristics and environmental properties in incremental, online, model-free reinforcement learning. We first identified six properties of environments that contribute to the difficulty of exploration. We propose that environments with high-variance, misleading, or sparse rewards are more difficult to explore, as well as environments with high-variance transitions, antagonistic transitions, or large state-action spaces. Exploration methods cannot be useful across a broad set of environments unless they appropriately address these properties. We hope that these properties will help guide researchers to develop general exploration methods for reinforcement learning.

We also propose a categorization of exploration methods according to their underlying exploration strategies. We group methods that behave optimistically with respect to a plausible value function into two groups: methods that select actions according to upper bounds are Optimistic in the Face of Uncertainty, and methods that use Thompson Sampling. Exploration in optimistic methods typically decays according to $\frac{1}{\sqrt{n}}$ in the number of samples observed. We propose a category of Learning Progress methods, which explore based on the learning progress of an auxiliary function or an initially inflated value function. Exploration in these methods typically decays by the exponent of the learning rate of the auxiliary function raised to the number of samples observed. We also identify a group of Noisy methods, which add a fixed or learned amount of noise to the value function or action-selection

mechanism. These categories add structure to the large body of work describing new exploration methods, and offers insight into the potential refinement of these methods in the future.

Finally, we conduct an empirical study examining how well the policies learned by each exploration method perform in environments based on each of the six exploration properties we identified previously. We find that none of the exploration methods perform very well on all the environments, and that they tend to learn better policies in some environments than others. Overall, L-UCLS (OFU) had the best performance across all environments. Interestingly, performance was usually grouped according to methods that used a reward bonus, took actions according to inflated values, or used randomness to explore. Since algorithmic mechanisms seemed to impact behaviour more than the exploration heuristics used, we suggest that assessing and designing different mechanisms to implement a single exploration heuristic may be a more appropriate direction for future work than designing new exploration heuristics for the RL setting.

## 7.1 Future Work

There are several interesting directions for future work. This thesis focused on Q-learning as the base RL algorithm, but algorithms that do planning updates, such as DynaQ or DQN, could also be studied in conjunction with exploration methods. We expect planning updates and other algorithmic features to interact differently with the exploration methods than Q-learning did in our experiments.

Future work could also focus on how exploration heuristics interact with fixed or learned function approximation schemes, which for example might drastically change the meaning of a state visitation count over time. Exploration methods built on top of deep RL methods like DQN are already affected by this interaction, which may have further implications for learning beyond exploration.

The experiments in this thesis could be further extended to understand how exploration methods balance performance and exploration during the training phase, as RL algorithms typically learn forever to continually adapt to small changes in applied environments. We plan to explore this direction in future work.

The statistical tests used in this work is typically absent from other empirical studies demonstrating the performance of exploration methods in RL. In fact, statistical tests are absent from almost all empirical work in RL. Experiment design in RL is frequently a complete random block design with replicates, which not very common in the statistical literature due to the relative expense of conducting so many experiments, there are two statistical directions that could extend this work.

The first is to design more efficient testing procedures for identifying which algorithms perform better than others in a set. While identifying high-performing algorithms is sometimes important, we do not consider it to be the purpose of algorithmic research in RL.

The second direction is to change the way RL experiments are conducted to make more efficient use of each trial. This efficiency may help with the evaluation of deep RL algorithms, which can be quite expensive to run. Deep RL experiments are sometimes reported with only a single trial per algorithm, which reduces the reliability of the results. Improved statistical testing methodology may help researchers carefully evaluate such computationally expensive algorithms.

Part of the difficulty with applying RL algorithms is that their hyperparameters must be tuned to ensure good performance. We saw in our experiments that the parameter sensitivity of most exploration methods is quite poor, somewhat shifting the computational burden from exploration during operation to hyperparameter tuning before operation. While this shift is beneficial in some applications, developing exploration methods that are robust to hyperparameter settings remains a critical area of research.

Finally, the experiments presented in this thesis only consider environ-

ments that are based on a single challenging exploration property. As there are 63 ($2^6 - 1$) combinations of these six properties, we leave the empirical investigation of their interactions to future work. Since the exploration properties were designed to be as orthogonal as possible, we expect that combining these properties will produce interesting and difficult testbeds for future exploration research.

# Bibliography


Asadi, Kavosh and Michael L Littman (2017). "An alternative softmax operator for reinforcement learning." In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, pp. 243–252. 26

Azar, Mohammad Gheshlaghi, Vicenç Gómez, and Hilbert J Kappen (2012). "Dynamic policy programming." In: *Journal of Machine Learning Research* 13.Nov, pp. 3207–3245. 26

Bellemare, Marc, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos (2016). "Unifying count-based exploration and intrinsic motivation." In: *Advances in Neural Information Processing Systems*, pp. 1471–1479. 28, 29

Bhandari, Jalaj and Daniel Russo (2019). "Global Optimality Guarantees For Policy Gradient Methods." In: *arXiv preprint 1906.01786*. 70

Burda, Yuri, Harrison Edwards, Amos Storkey, and Oleg Klimov (2018). "Exploration by random network distillation." In: *arXiv preprint 1810.12894*. 33

Choshen, Leshem, Lior Fox, and Yonatan Loewenstein (2018). "Dora the explorer: Directed outreaching reinforcement action-selection." In: *arXiv preprint 1804.04012*. 30

Dayan, Peter (1993). "Improving generalization for temporal difference learning: The successor representation." In: *Neural Computation* 5.4, pp. 613–624. 30

De Asis, Kristopher, J Fernando Hernandez-Garcia, G Zacharias Holland, and Richard S Sutton (2018). "Multi-step reinforcement learning: A unifying algorithm." In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 11

Degris, Thomas, Martha White, and Richard S Sutton (2012). "Linear off-policy actor-critic." In: *In International Conference on Machine Learning*. Citeseer. 26

Du, Simon S and Sham M Kakade (2019). "Is a Good Representation Sufficient for Sample Efficient Reinforcement Learning?" en. In: p. 17. 20, 24, 43

Duncan, Luce R (1959). *Individual choice behavior: A theoretical analysis*. 26

Facebook (2020). *ReAgent: Applied Reinforcement Learning Platform*. URL: `https://reagent.ai` (visited on 01/08/2020). 1

Fortunato, Meire, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier

Pietquin, Charles Blundell, and Shane Legg (2018). “Noisy networks for exploration.” In: *International Conference on Learning Representations*. 35

Gal, Yarin and Zoubin Ghahramani (2016). “Dropout as a bayesian approximation: Representing model uncertainty in deep learning.” In: *international conference on machine learning*, pp. 1050–1059. 31

Gutenbrunner, Christoph, JKRS Jurečková, Roger Koenker, and Stephen Portnoy (1993). “Tests of linear hypotheses based on regression rank scores.” In: *Journal of Nonparametric Statistics* 2.4, pp. 307–331. 69

Haarnoja, Tuomas, Haoran Tang, Pieter Abbeel, and Sergey Levine (2017). “Reinforcement learning with deep energy-based policies.” In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, pp. 1352–1361. 26

Hernandez-Garcia, J Fernando and Richard S Sutton (2019). “Understanding Multi-Step Deep Reinforcement Learning: A Systematic Study of the DQN Target.” In: *arXiv preprint 1901.07510*. 11

Koenker, Roger (2019). *quantreg: Quantile Regression*. R package version 5.54. URL: https://CRAN.R-project.org/package=quantreg. 69

Kozuno, Tadashi, Eiji Uchibe, and Kenji Doya (2019). “Theoretical Analysis of Efficiency and Robustness of Softmax and Gap-Increasing Operators in Reinforcement Learning.” In: *The 22nd International Conference on Artificial Intelligence and Statistics*, pp. 2995–3003. 26

Kumaraswamy, Raksha, Matthew Schlegel, Adam White, and Martha White (2018). “Context-dependent upper-confidence bounds for directed exploration.” In: *Advances in Neural Information Processing Systems*, pp. 4779–4789. 4, 30, 35

Lagoudakis, Michail G and Ronald Parr (2003). “Least-squares policy iteration.” In: *Journal of machine learning research* 4.Dec, pp. 1107–1149. 11

Lai, Tze Leung and Herbert Robbins (1985). “Asymptotically efficient adaptive allocation rules.” In: *Advances in applied mathematics* 6.1, pp. 4–22. 29

Langford, John (2018). *RL Acid*. Retrieved from https://github.com/JohnLangford/RL_acid. 3, 15, 20, 39, 41

Lattimore, Tor and Csaba Szepesvari (2019). “Learning with Good Feature Representations in Bandits and in RL with a Generative Model.” In: *arXiv preprint arXiv:1911.07676*. 24, 25, 43

Machado, Marlos C, Marc G Bellemare, and Michael Bowling (2018). “Count-based exploration with the successor representation.” In: *arXiv preprint 1807.11622*. 30

Machado, Marlos C, Sriram Srinivasan, and Michael Bowling (2015). “Domain-independent optimistic initialization for reinforcement learning.” In: *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*. 35

Martin, Jarryd, Suraj Narayanan Sasikumar, Tom Everitt, and Marcus Hutter (2017). “Count-based exploration in feature space for reinforcement learning.” In: *arXiv preprint 1706.08090*. 30, 43

Melo, Francisco S (2001). *Convergence of Q-learning: A simple proof*. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.413.2350&rep=rep1&type=pdf (visited on 01/12/2020). 35

Melo, Francisco S, Sean P Meyn, and M Isabel Ribeiro (2008). "An analysis of reinforcement learning with function approximation." In: *Proceedings of the 25th international conference on Machine learning*. ACM. 10

MEULEAU, NICOLAS (1999). "Exploration of Multi-State Environments: Local Measures and Back-Propagation of Uncertainty." en. In: p. 38. 29

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. (2015). "Human-level control through deep reinforcement learning." In: *Nature* 518.7540, p. 529. 11, 29, 42

Moore, Andrew William (1990). "Efficient memory-based learning for robot control." en. In: p. 248. 16, 40

Nachum, Ofir, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans (2017). "Bridging the gap between value and policy based reinforcement learning." In: *Advances in Neural Information Processing Systems*, pp. 2775–2785. 26

Osband, Ian, John Aslanides, and Albin Cassirer (2018). "Randomized prior functions for deep reinforcement learning." In: *Advances in Neural Information Processing Systems*, pp. 8617–8629. 31

Osband, Ian, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy (2016). "Deep exploration via bootstrapped DQN." In: *Advances in neural information processing systems*, pp. 4026–4034. 31

Osband, Ian, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepezvari, Satinder Singh, Benjamin Van Roy, Richard Sutton, David Silver, and Hado van Hasselt (2019). "Behaviour suite for reinforcement learning." In: *arXiv preprint 1908.03568*. 3

Osband, Ian, Benjamin Van Roy, Daniel J. Russo, and Zheng Wen (2019). "Deep Exploration via Randomized Value Functions." In: *Journal of Machine Learning Research* 20.124, pp. 1–62. 20, 35

Pathak, Deepak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell (2017). "Curiosity-driven exploration by self-supervised prediction." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 16–17. 32

Peters, Jan, Katharina Mulling, and Yasemin Altun (2010). "Relative entropy policy search." In: *Twenty-Fourth AAAI Conference on Artificial Intelligence*. 26

Plappert, Matthias, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz (2018). "Parameter space noise for exploration." In: *International Conference on Learning Representations*. 36

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria. URL: https://www.R-project.org/. 58

Russo, Daniel (2019). *Algorithmic Foundations of Learning and Control*. URL: https://www.youtube.com/watch?v=qqTMnQ-NLU (visited on 12/18/2019). 41

Russo, Daniel J., Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen (2018). "A Tutorial on Thompson Sampling." In: *Foundations and Trends® in Machine Learning* 11.1, pp. 1–96. 27

Services, Amazon Web (2020). *Use Reinforcement Learning with Amazon SageMaker*. URL: https://docs.aws.amazon.com/sagemaker/latest/dg/reinforcement-learning.html (visited on 01/08/2020). 1

Silver, David, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. (2017). "Mastering chess and shogi by self-play with a general reinforcement learning algorithm." In: *arXiv:1712.01815*. 1

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2014). "Dropout: a simple way to prevent neural networks from overfitting." In: *The journal of machine learning research* 15.1, pp. 1929–1958. 31

Stadie, Bradly C, Sergey Levine, and Pieter Abbeel (2015). "Incentivizing exploration in reinforcement learning with deep predictive models." In: *arXiv preprint 1507.00814*. 32

Still, Susanne (2009). "Information-theoretic approach to interactive learning." In: *EPL (Europhysics Letters)* 85.2, p. 28005. 26

Strehl, Alexander L. and Michael L. Littman (2008). "An analysis of model-based Interval Estimation for Markov Decision Processes." In: *Journal of Computer and System Sciences* 74.8, pp. 1309–1331. 23, 29

Sutton, Richard S (1991). "Dyna, an integrated architecture for learning, planning, and reacting." In: *ACM Sigart Bulletin*. 11

Sutton, Richard S and Andrew G Barto (2018). *Reinforcement learning: An introduction*. MIT press. 9, 17, 21, 40, 55

Tang, Haoran, Rein Houthooft, Davis Foote, Adam Stooke, OpenAI Xi Chen, Yan Duan, John Schulman, Filip DeTurck, and Pieter Abbeel (2017). "# Exploration: A study of count-based exploration for deep reinforcement learning." In: *Advances in neural information processing systems*, pp. 2753–2762. 30

Tange, Ole (2018). *GNU Parallel 2018*. Ole Tange. ISBN: 9781387509881. DOI: 10.5281/zenodo.1146014. URL: https://doi.org/10.5281/zenodo.1146014.

Thompson, William R (1933). "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples." In: *Biometrika* 25.3/4, pp. 285–294. 27

Thrun, Sebastian B (1992). "The role of exploration in learning control." In: *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, pp. 1–27. 35

Tsitsiklis, John N (1994). "Asynchronous stochastic approximation and Q-learning." In: *Machine learning* 16.3, pp. 185–202. 35

Van Roy, Benjamin and Shi Dong (2019). "Comments on the Du-Kakade-Wang-Yang Lower Bounds." In: *arXiv preprint arXiv:1911.07910*. 24, 25, 43

van Seijen, Harm and Rich Sutton (2015). "A deeper look at planning as learning from replay." In: *International conference on machine learning*, pp. 2314–2322. 11

Watkins, Christopher John Cornish Hellaby (1989). "Learning from delayed rewards." PhD thesis. King's College, Cambridge. 9

White, Martha and Adam White (2010). "Interval Estimation for Reinforcement-Learning Algorithms in Continuous-State Domains." In: *Advances in Neural Information Processing Systems 23*, pp. 2433–2441. 14, 39

Wilke, Claus O. (2018). *ggridges: Ridgeline Plots in 'ggplot2'*. R package version 0.5.1. URL: https://CRAN.R-project.org/package=ggridges. 58

Zou, Shaofeng, Tengyu Xu, and Yingbin Liang (2019). "Finite-sample analysis for sarsa with linear function approximation." In: *Advances in Neural Information Processing Systems*. 10

# Appendix A

# Hyperparameter settings

Back to Chapter 6
Back to Table of Contents

Here we report the best hyperparameter settings in each of the environments from Chapter 3. They were identified by running each exploration method 72 times and selecting the settings with the highest median accumulated reward in the testing phase. If two agents were tied for the highest median, we selected the settings that produced the highest mean accumulated reward in the testing phase. We include the median performance of each method for reference.

| Method | $\alpha$ | $\alpha_2$ | $\beta$ | $\theta$ | $\sigma_{\max}$ | $\epsilon$ | $\mathbf{w}_{\text{bonus}}$ | Tiles | Tilings | Median |
|---|---|---|---|---|---|---|---|---|---|---|
| S-counts | $2^{-4}$ | | 0.005 | | | | | 9 | 10 | 0.00 |
| SA-counts | $2^{-5}$ | | 0.100 | | | | | 2 | 32 | 96.33 |
| BootstrapQ | $2^{-3}$ | | | | | | | 2 | 32 | 104.53 |
| $\epsilon$-greedy | $2^{-3}$ | | | | | $2^{-1}$ | | 48 | 2 | 380.75 |
| IEQL+ | $2^{-5}$ | | | 0.99 | 0.010 | | | 2 | 32 | 104.49 |
| NoisyNets | $2^{-4}$ | | | | | | | 48 | 2 | 0.00 |
| OptInit | $2^{-5}$ | | | | | | 50 | 2 | 32 | 5502.30 |
| RND-S | $2^{-3}$ | | 0.005 | | | | | 2 | 32 | 0.00 |
| RND-SA | $2^{-3}$ | | 0.005 | | | | | 2 | 32 | 0.00 |
| SoftmaxAC | $2^{-5}$ | 0.031 | | | | | | 9 | 10 | 9926.50 |
| L-UCLS (OFU) | $2^{-5}$ | 0.500 | | 0.01 | | | | 48 | 2 | 5803.25 |
| L-UCLS (TS) | $2^{-5}$ | 0.050 | | Inf | | | | 48 | 2 | 2568.56 |

Table A.1: Best hyperparameter settings in VarianceWorld

| Method | $\alpha$ | $\alpha_2$ | $\beta$ | $\theta$ | $\sigma_{\max}$ | $\epsilon$ | $\mathbf{w}_{\text{bonus}}$ | Tiles | Tilings | Median |
|---|---|---|---|---|---|---|---|---|---|---|
| S-counts | $2^{-5}$ | | 0.005 | | | | | 9 | 10 | 611.50 |
| SA-counts | $2^{-3}$ | | 0.050 | | | | | 2 | 32 | 2253.50 |
| BootstrapQ | $2^{-5}$ | | | | | | | 9 | 10 | 1749.50 |
| $\epsilon$-greedy | $2^{-5}$ | | | | | $2^{-6}$ | | 9 | 10 | 3394.75 |
| IEQL+ | $2^{-2}$ | | | 0.90 | 0.015 | | | 2 | 32 | 2665.25 |
| NoisyNets | $2^{-4}$ | | | | | | | 9 | 10 | 0.00 |
| OptInit | $2^{-5}$ | | | | | | 1 | 2 | 32 | 3143.00 |
| RND-S | $2^{-5}$ | | 0.010 | | | | | 9 | 10 | 555.50 |
| RND-SA | $2^{-5}$ | | 0.005 | | | | | 9 | 10 | 617.00 |
| SoftmaxAC | $2^{-4}$ | 0.250 | | | | | | 48 | 2 | 3834.00 |
| L-UCLS (OFU) | $2^{-5}$ | 0.005 | | 0.04 | | | | 9 | 10 | 3518.25 |
| L-UCLS (TS) | $2^{-5}$ | 0.005 | | -1.33 | | | | 9 | 10 | 3461.50 |

Table A.2: Best hyperparameter settings in WindyJump

| Method | $\alpha$ | $\alpha_2$ | $\beta$ | $\theta$ | $\sigma_{\max}$ | $\epsilon$ | $\mathbf{w}_{\text{bonus}}$ | Tiles | Tilings | Median |
|---|---|---|---|---|---|---|---|---|---|---|
| S-counts | $2^{-4}$ | | 0.100 | | | | | 2 | 32 | 3712.11 |
| SA-counts | $2^{-2}$ | | 0.100 | | | | | 9 | 10 | 3712.11 |
| BootstrapQ | $2^{-4}$ | | | | | | | 9 | 10 | 5.49 |
| $\epsilon$-greedy | $2^{-2}$ | | | | | $2^{-1}$ | | 48 | 2 | 5.49 |
| IEQL+ | $2^{-3}$ | | | 0.75 | 0.005 | | | 9 | 10 | 3712.11 |
| NoisyNets | $2^{-5}$ | | | | | | | 48 | 2 | 5.46 |
| OptInit | $2^{-3}$ | | | | | | 1 | 48 | 2 | 3712.11 |
| RND-S | $2^{-5}$ | | 0.100 | | | | | 2 | 32 | 3712.11 |
| RND-SA | $2^{-3}$ | | 0.010 | | | | | 9 | 10 | 3712.11 |
| SoftmaxAC | 1 | 0.062 | | | | | | 2 | 32 | 3704.66 |
| L-UCLS (OFU) | 1 | 0.100 | | -1.33 | | | | 48 | 2 | 3712.11 |
| L-UCLS (TS) | $2^{-1}$ | 0.500 | | -1.00 | | | | 48 | 2 | 3712.11 |

Table A.3: Best hyperparameter settings in Antishaping

| Method | $\alpha$ | $\alpha_2$ | $\beta$ | $\theta$ | $\sigma_{\max}$ | $\epsilon$ | $\mathbf{w}_{\text{bonus}}$ | Tiles | Tilings | Median |
|---|---|---|---|---|---|---|---|---|---|---|
| S-counts | $2^{-3}$ | | 0.010 | | | | | 9 | 10 | 5923.66 |
| SA-counts | $2^{-3}$ | | 0.005 | | | | | 9 | 10 | 5921.82 |
| BootstrapQ | 1 | | | | | | | 2 | 32 | 1000.00 |
| $\epsilon$-greedy | 1 | | | | | $2^{-4}$ | | 9 | 10 | 1000.00 |
| IEQL+ | $2^{-3}$ | | | 0.99 | 0.015 | | | 2 | 32 | 5937.11 |
| NoisyNets | $2^{-5}$ | | | | | | | 9 | 10 | 5356.81 |
| OptInit | $2^{-3}$ | | | | | | 50 | 48 | 2 | 5908.99 |
| RND-S | $2^{-4}$ | | 0.050 | | | | | 2 | 32 | 5934.66 |
| RND-SA | $2^{-5}$ | | 0.010 | | | | | 2 | 32 | 5934.05 |
| SoftmaxAC | $2^{-5}$ | 1.000 | | | | | | 48 | 2 | 997.00 |
| L-UCLS (OFU) | $2^{-4}$ | 0.100 | | -1.00 | | | | 48 | 2 | 5908.99 |
| L-UCLS (TS) | $2^{-1}$ | 0.500 | | 0.01 | | | | 2 | 32 | 1000.00 |

Table A.4: Best hyperparameter settings in AlpineSki

| Method | $\alpha$ | $\alpha_2$ | $\beta$ | $\theta$ | $\sigma_{\max}$ | $\epsilon$ | $\mathbf{w}_{\text{bonus}}$ | Tiles | Tilings | Median |
|---|---|---|---|---|---|---|---|---|---|---|
| S-counts | $2^{-2}$ | | 0.005 | | | | | 16 | 2 | 0.00 |
| SA-counts | $2^{-2}$ | | 0.005 | | | | | 16 | 2 | 43.42 |
| BootstrapQ | $2^{-4}$ | | | | | | | 8 | 7 | 2608.54 |
| $\epsilon$-greedy | $2^{-4}$ | | | | | $2^{-4}$ | | 8 | 7 | 2960.91 |
| IEQL+ | $2^{-2}$ | | | 0.50 | 0.010 | | | 16 | 2 | 101.87 |
| NoisyNets | $2^{-5}$ | | | | | | | 16 | 2 | 733.13 |
| OptInit | $2^{-1}$ | | | | | | 1 | 8 | 7 | 2852.36 |
| RND-S | 1 | | 0.010 | | | | | 8 | 7 | 13.36 |
| RND-SA | $2^{-3}$ | | 0.010 | | | | | 16 | 2 | 41.75 |
| SoftmaxAC | 1 | 0.500 | | | | | | 2 | 64 | 2895.78 |
| L-UCLS (OFU) | $2^{-3}$ | 0.100 | | -1.00 | | | | 8 | 7 | 2748.82 |
| L-UCLS (TS) | $2^{-2}$ | 0.005 | | -1.00 | | | | 8 | 7 | 2578.48 |

Table A.5: Best hyperparameter settings in Sparse MountainCar

| Method | $\alpha$ | $\alpha_2$ | $\beta$ | $\theta$ | $\sigma_{\max}$ | $\epsilon$ | $\mathbf{w}_{\text{bonus}}$ | Tiles | Tilings | Median |
|---|---|---|---|---|---|---|---|---|---|---|
| S-counts | $2^{-4}$ | | 0.010 | | | | | 2 | 54 | 899.67 |
| SA-counts | $2^{-4}$ | | 0.005 | | | | | 4 | 12 | 8.10 |
| BootstrapQ | $2^{-2}$ | | | | | | | 8 | 2 | 8.23 |
| $\epsilon$-greedy | $2^{-2}$ | | | | | $2^{-1}$ | | 8 | 2 | 4003.97 |
| IEQL+ | $2^{-3}$ | | | 0.99 | 0.015 | | | 4 | 12 | 8.10 |
| NoisyNets | $2^{-4}$ | | | | | | | 2 | 54 | 899.72 |
| OptInit | 1 | | | | | | 1 | 8 | 2 | 8.23 |
| RND-S | $2^{-3}$ | | 0.005 | | | | | 4 | 12 | 8.10 |
| RND-SA | $2^{-2}$ | | 0.005 | | | | | 4 | 12 | 8.10 |
| SoftmaxAC | 1 | 1.000 | | | | | | 2 | 54 | 4054.36 |
| L-UCLS (OFU) | $2^{-2}$ | 0.010 | | 0.01 | | | | 8 | 2 | 2242.48 |
| L-UCLS (TS) | $2^{-4}$ | 0.050 | | -1.00 | | | | 2 | 54 | 3777.05 |

Table A.6: Best hyperparameter settings in Hypercube

# Appendix B

# Results for all Hyperparameter Settings

Back to Parameter Study
Back to Table of Contents

Here we present the density plots of the performance of each exploration method, across all hyperparameter settings.

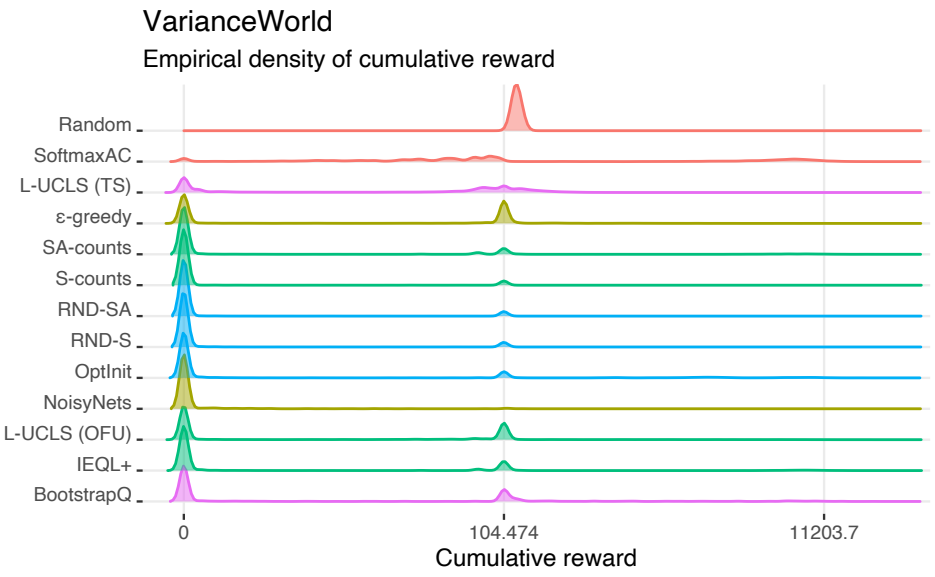


Figure B.1: This plot shows a distribution over the reward accumulated by each agent over the 100000 time-step test phase, for all hyperparameter settings. Agents are colored according to their underlying heuristics, and sorted by their median performance. We calculate the plotted densities using the `ggridges` package for `R`, and data from 72 independent runs for each hyperparameter setting.

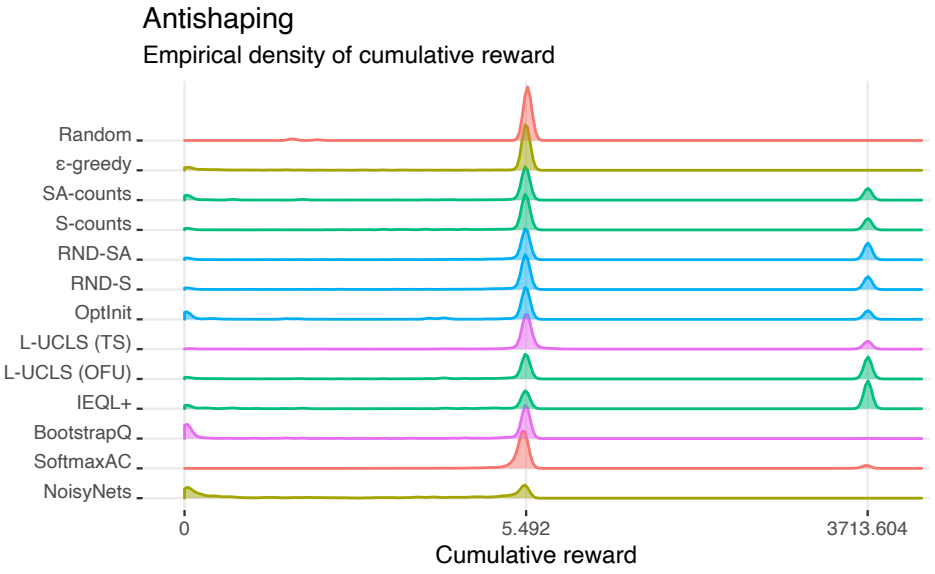


Figure B.2: This plot shows a distribution over the reward accumulated by each agent over the 100000 time-step test phase, for all hyperparameter settings. Agents are colored according to their underlying heuristics, and sorted by their median performance. We calculate the plotted densities using the `ggridges` package for `R`, and data from 72 independent runs for each hyperparameter setting.

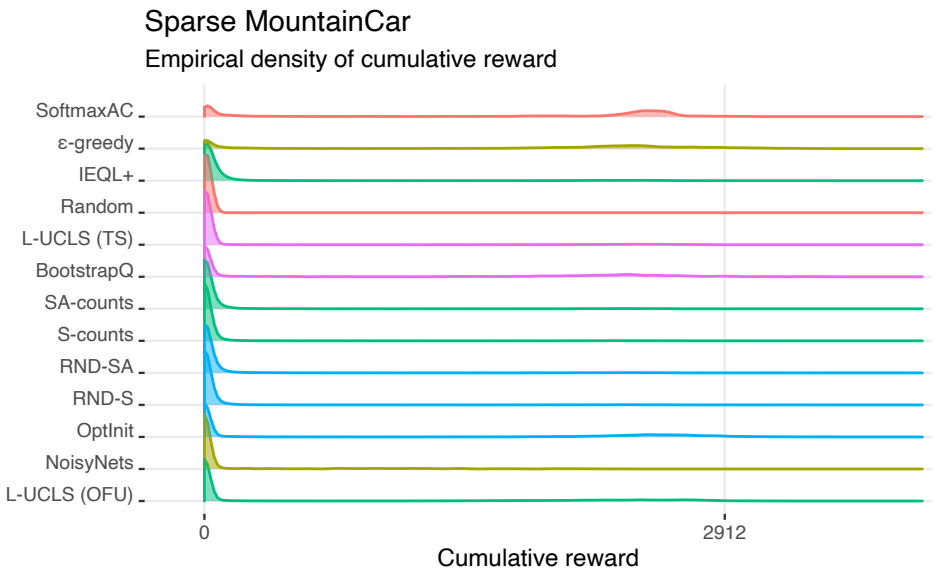


Figure B.3: This plot shows a distribution over the reward accumulated by each agent over the 100000 time-step test phase, for all hyperparameter settings. Agents are colored according to their underlying heuristics, and sorted by their median performance. We calculate the plotted densities using the `ggridges` package for `R`, and data from 72 independent runs for each hyperparameter setting.

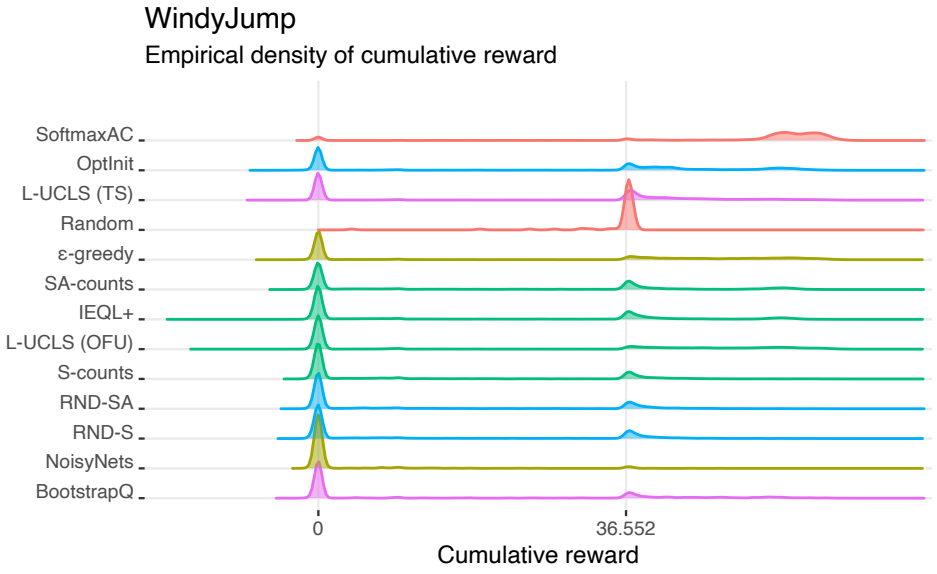


Figure B.4: This plot shows a distribution over the reward accumulated by each agent over the 100000 time-step test phase, for all hyperparameter settings. Agents are colored according to their underlying heuristics, and sorted by their median performance. We calculate the plotted densities using the `ggridges` package for `R`, and data from 72 independent runs for each hyperparameter setting.

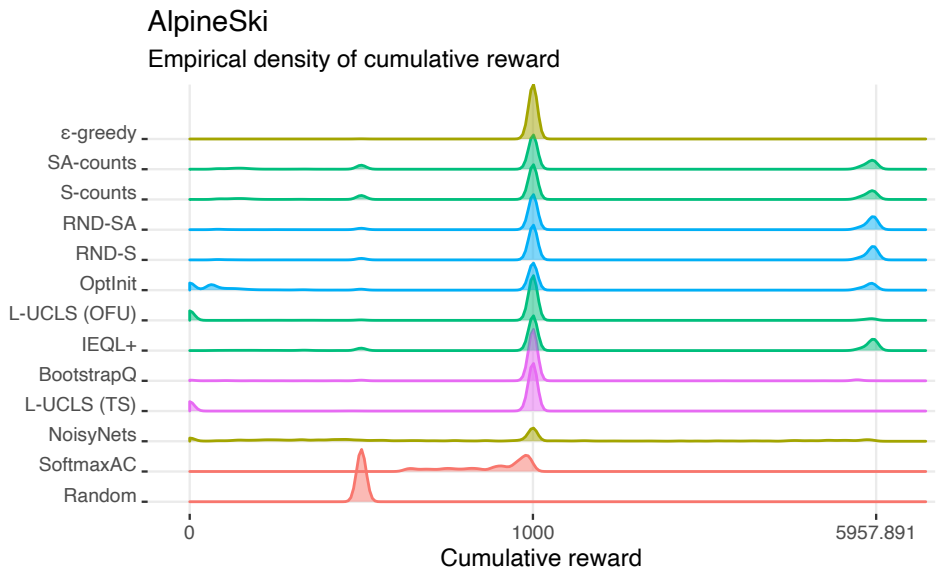


Figure B.5: This plot shows a distribution over the reward accumulated by each agent over the 100000 time-step test phase, for all hyperparameter settings. Agents are colored according to their underlying heuristics, and sorted by their median performance. We calculate the plotted densities using the `ggridges` package for `R`, and data from 72 independent runs for each hyperparameter setting.

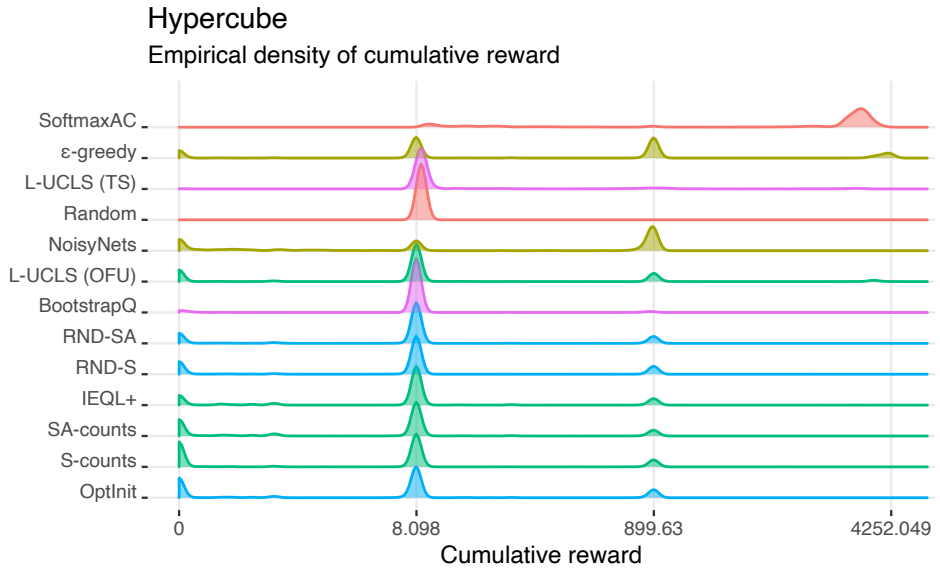


Figure B.6: This plot shows a distribution over the reward accumulated by each agent over the 100000 time-step test phase, for all hyperparameter settings. Agents are colored according to their underlying heuristics, and sorted by their median performance. We calculate the plotted densities using the `ggridges` package for `R`, and data from 72 independent runs for each hyperparameter setting.

# Appendix C

# Quantile Regression Table

Back to Statistical Analysis

Back to Table of Contents

Here we present the summary tables detailing the results of Student T-tests on the significance of terms in our median regression model.

Table C.1: Output of the call `summary(rq(value agent env, tau=0.5), se='ker')`, using the `quantreg` package in `R`.

| Variable | Value | Std. Error | t value | Pr(>\|t\|) |
|---|---|---|---|---|
| Intercept | 5908.99 | 1.30 | 4542.49 | 0.00 |
| agentbellemare_agent_Q | 14.67 | 2.05 | 7.15 | 0.00 |
| agentbellemare_state_action_Q | 12.23 | 2.44 | 5.00 | 0.00 |
| agentbootstrap_Q | -4908.99 | 1.53 | -3208.99 | 0.00 |
| agentepsilon_greedy_Q | -4908.99 | 1.57 | -3128.44 | 0.00 |
| agentIEQL+_Q | 28.12 | 2.11 | 13.32 | 0.00 |
| agentnoisy_net_Q | -560.37 | 18.10 | -30.96 | 0.00 |
| agentoptimistic_init_Q | 0.00 | 1.76 | 0.00 | 1.00 |
| agentrandom | -5409.08 | 1.45 | -3725.37 | 0.00 |
| agentRND_state | 25.68 | 2.96 | 8.66 | 0.00 |
| agentRND_state_action | 24.45 | 3.24 | 7.56 | 0.00 |
| agentsoftmax_AC | -4911.99 | 1.57 | -3129.21 | 0.00 |
| agentUCLS_TS | -4908.99 | 1.57 | -3128.43 | 0.00 |
| envSparseMC | -3156.83 | 13.04 | -242.09 | 0.00 |
| envVarianceWorld | -128.99 | 63.20 | -2.04 | 0.04 |
| envHypercube | -2922.27 | 63.20 | -46.24 | 0.00 |
| envAntishaping | -2196.87 | 1.67 | -1318.24 | 0.00 |
| envWindyJump | -2392.99 | 27.49 | -87.06 | 0.00 |
| agentbellemare_agent_Q:envSparseMC | -2766.83 | 13.22 | -209.30 | 0.00 |

| | | | | |
|---|---|---|---|---|
| agentbellemare_state_action_Q:envSparseMC | -2720.97 | 19.34 | -140.72 | 0.00 |
| agentbootstrap_Q:envSparseMC | 4752.01 | 28.14 | 168.86 | 0.00 |
| agentepsilon_greedy_Q:envSparseMC | 5109.39 | 33.86 | 150.89 | 0.00 |
| agentIEQL+_Q:envSparseMC | -2676.74 | 21.47 | -124.68 | 0.00 |
| agentnoisy_net_Q:envSparseMC | -1450.31 | 31.57 | -45.95 | 0.00 |
| agentoptimistic_init_Q:envSparseMC | 100.20 | 17.84 | 5.62 | 0.00 |
| agentrandom:envSparseMC | 2663.60 | 13.08 | 203.62 | 0.00 |
| agentRND_state:envSparseMC | -2764.48 | 13.72 | -201.44 | 0.00 |
| agentRND_state_action:envSparseMC | -2733.19 | 17.72 | -154.25 | 0.00 |
| agentsoftmax_AC:envSparseMC | 5055.61 | 15.26 | 331.20 | 0.00 |
| agentUCLS_TS:envSparseMC | 4731.97 | 36.40 | 129.99 | 0.00 |
| agentbellemare_agent_Q:envVarianceWorld | -5794.67 | 63.22 | -91.65 | 0.00 |
| agentbellemare_state_action_Q:envVarianceWorld | -5695.94 | 63.34 | -89.92 | 0.00 |
| agentbootstrap_Q:envVarianceWorld | -766.48 | 63.22 | -12.12 | 0.00 |
| agentepsilon_greedy_Q:envVarianceWorld | -506.01 | 89.37 | -5.66 | 0.00 |
| agentIEQL+_Q:envVarianceWorld | -5703.63 | 63.24 | -90.18 | 0.00 |
| agentnoisy_net_Q:envVarianceWorld | -5219.63 | 65.73 | -79.41 | 0.00 |
| agentoptimistic_init_Q:envVarianceWorld | -286.27 | 88.69 | -3.23 | 0.00 |
| agentrandom:envVarianceWorld | 160.08 | 63.68 | 2.51 | 0.01 |
| agentRND_state:envVarianceWorld | -5805.68 | 63.27 | -91.77 | 0.00 |
| agentRND_state_action:envVarianceWorld | -5804.45 | 63.28 | -91.73 | 0.00 |
| agentsoftmax_AC:envVarianceWorld | 9050.99 | 87.61 | 103.30 | 0.00 |
| agentUCLS_TS:envVarianceWorld | 1671.41 | 71.11 | 23.50 | 0.00 |
| agentbellemare_agent_Q:envHypercube | -2101.74 | 63.24 | -33.24 | 0.00 |
| agentbellemare_state_action_Q:envHypercube | -2990.84 | 63.24 | -47.30 | 0.00 |
| agentbootstrap_Q:envHypercube | 1930.50 | 63.21 | 30.54 | 0.00 |
| agentepsilon_greedy_Q:envHypercube | 5909.40 | 66.97 | 88.24 | 0.00 |
| agentIEQL+_Q:envHypercube | -3006.74 | 63.22 | -47.56 | 0.00 |
| agentnoisy_net_Q:envHypercube | -1526.63 | 65.73 | -23.23 | 0.00 |
| agentoptimistic_init_Q:envHypercube | -2978.47 | 63.22 | -47.11 | 0.00 |
| agentrandom:envHypercube | 2449.10 | 63.20 | 38.75 | 0.00 |
| agentRND_state:envHypercube | -3004.29 | 63.26 | -47.49 | 0.00 |
| agentRND_state_action:envHypercube | -3003.07 | 63.28 | -47.46 | 0.00 |
| agentsoftmax_AC:envHypercube | 5981.67 | 63.75 | 93.83 | 0.00 |
| agentUCLS_TS:envHypercube | 5696.86 | 64.63 | 88.14 | 0.00 |
| agentbellemare_agent_Q:envAntishaping | -14.67 | 2.92 | -5.02 | 0.00 |
| agentbellemare_state_action_Q:envAntishaping | -12.23 | 2.96 | -4.13 | 0.00 |
| agentbootstrap_Q:envAntishaping | 1202.36 | 2.14 | 560.57 | 0.00 |
| agentepsilon_greedy_Q:envAntishaping | 1202.36 | 2.08 | 578.66 | 0.00 |
| agentIEQL+_Q:envAntishaping | -28.12 | 2.54 | -11.05 | 0.00 |
| agentnoisy_net_Q:envAntishaping | -3146.28 | 18.14 | -173.48 | 0.00 |
| agentoptimistic_init_Q:envAntishaping | -0.00 | 2.29 | -0.00 | 1.00 |
| agentrandom:envAntishaping | 1714.36 | 2.12 | 810.45 | 0.00 |
| agentRND_state:envAntishaping | -25.68 | 3.54 | -7.25 | 0.00 |
| agentRND_state_action:envAntishaping | -24.45 | 3.62 | -6.75 | 0.00 |

| | | | | |
|---|---|---|---|---|
| agentsoftmax_AC:envAntishaping | 4904.53 | 2.55 | 1920.40 | 0.00 |
| agentUCLS_TS:envAntishaping | 4908.99 | 2.14 | 2298.47 | 0.00 |
| agentbellemare_agent_Q:envWindyJump | -2930.17 | 55.21 | -53.07 | 0.00 |
| agentbellemare_state_action_Q:envWindyJump | -1370.23 | 68.93 | -19.88 | 0.00 |
| agentbootstrap_Q:envWindyJump | 3077.99 | 66.11 | 46.56 | 0.00 |
| agentepsilon_greedy_Q:envWindyJump | 4771.49 | 42.27 | 112.89 | 0.00 |
| agentIEQL+_Q:envWindyJump | -908.12 | 68.92 | -13.18 | 0.00 |
| agentnoisy_net_Q:envWindyJump | -2955.63 | 32.90 | -89.84 | 0.00 |
| agentoptimistic_init_Q:envWindyJump | -369.00 | 36.25 | -10.18 | 0.00 |
| agentrandom:envWindyJump | 1976.10 | 28.09 | 70.35 | 0.00 |
| agentRND_state:envWindyJump | -2979.68 | 64.15 | -46.45 | 0.00 |
| agentRND_state_action:envWindyJump | -2937.95 | 68.97 | -42.60 | 0.00 |
| agentsoftmax_AC:envWindyJump | 5229.49 | 31.10 | 168.17 | 0.00 |
| agentUCLS_TS:envWindyJump | 4849.99 | 41.66 | 116.42 | 0.00 |