Value Bonuses Using Ensemble Errors For Exploration in Reinforcement Learning

by

Abdul Wahab

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

Optimistic value estimates provide one mechanism for directed exploration in reinforcement learning (RL). The agent acts greedily with respect to an estimate of the value plus what can be seen as a *value bonus*. The value bonus can be learned by estimating a value function on *reward bonuses*, propagating local uncertainties around rewards. This approach, however, only increases the value bonus for an action retroactively, after seeing a higher reward bonus from that state and action. Such an approach does not encourage the agent to visit a state and action for the first time. In this work, we introduce an algorithm for exploration called Value Bonuses with Ensemble errors (VBE), that maintains an ensemble of random action-value functions (RQFs). VBE uses the errors in the estimation of these RQFs for designing value bonuses that provide first-visit optimism and deep exploration. The key idea is to design the rewards for these RQFs in such a way that the value bonus can decrease to zero. We show that VBE outperforms Bootstrap DQN and two reward bonus approaches (RND and ACB) on several classic environments used to test exploration and provide demonstrative experiments that it learns faster in several Atari environments.

Preface

This work was carried out in conjunction with Raksha Kumaraswamy and Martha White. This work is currently under review for publication. Due to the collaborative nature of this work, pronoun "we" is used in this script. However, I remain solely responsible for any technical or presentational errors present.

Acknowledgements

I am grateful to my supervisor Martha for allowing me the opportunity to work with her. It has truly been an honor and a privilege for me. I have learned many lessons from her for which I am very grateful. I owe my deepest gratitude to Raksha who acted as my co-supervisor and mentor for the entire duration of this work. I thank both of you for being so very patient with me and for your consistent support and help.

I would also like to thank my friends Saqib and Gohar for their support and for making my time here more enjoyable. I would like to thank Callie for constantly reminding me to focus on writing and not to procrastinate. I thank Shivam for teaching me to overcome the fear of sounding stupid and to ask questions. I am also thankful to all my other friends and the wonderful people here at UofA. It is a privilege to be among such an amazing and caring group of people.

Lastly, I would like to thank my parents and my sister for their immense support and love. I hope that I live up to your high expectations and be the person you all want me to be. Thank you!

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

An agent interacting with an environment for multiple steps, with the goal to maximize some scalar reward requires that the agent visits different parts of the environment, states, that can result in high reward. The goal of an agent is thus to understand the environment it is in, and perform a series of actions that can result in a high total reward. This series of state dependent actions that the agent takes is what we usually refer to as a policy. In order for an agent to find a policy that maximizes the total reward it receives, it needs to carefully explore the environment it is in. Exploration thus controls the degree of what can be known to the agent about the environment, and thus its ability to learn a policy that maximizes the reward. However, it is also not feasible for the agent to always explore the environment, as it also needs to exploit the knowledge it has gained to maximize the reward. The question then becomes a matter of time or instance, i.e., when should an agent explore and when should an agent exploit. This trade-off between exploration and exploitation is known as the exploration-exploitation dilemma.

There is a large body of literature that studies this dilemma, in the bandit setting and also in the full Reinforcement Learning (RL) setting. It is not very clear what is the optimal way to explore, or how an intelligent agent should explore an environment. However, in the tabula rasa setting where there is no prior information about the environment or the agent, the best or perhaps the only way to explore an environment, from the perspective of learning a policy that maximizes the total reward, is to try to visit all the unique states enough times to have accurate estimates for each state. One way of doing this is through a framework called Optimism in the Face of Uncertainty, which has been widely used and studied both in the bandit setting and the RL setting. The idea is that the agent should explore by taking actions that could be good, given the uncertainty (effectively an upper confidence bound).

The challenge with this approach is that it is not always easy to measure uncertainty, especially in the RL setting especially with function approximation. One practical way of doing this for the

tabular bandit setting is to maintain visitation counts for each action and define a bonus, often referred to as the *reward bonus*, using these counts. The agent then selects actions that maximizes the reward and this added reward bonus, encouraging the agent to take actions less frequently taken. The reward bonus can be defined in many different ways, and broadly such methods belong to a class of algorithms called Upper Confidence Bounds (UCB). Extending such UCB-style methods to the RL setting is not as straight forward, as the goal is not to maximize the immediate rewards, rather to maximize the expected sum of future rewards, i.e., the value of a state. Thus we need a bonus in the value space, which we refer to as the *value bonus*. This value bonus should be based on an uncertainty measure that does not just tell the agent about the immediate uncertainty of a state, but rather tells the uncertainty of future states as well, that the agent might observe while following a particular policy.

Exploration not only controls the ability of the agent to learn an optimal policy, but also the time it takes to learn that policy. If the agent explores randomly it will take a much longer time to cover the state space and learn a good policy. However, in many cases random exploration is the choice of exploration, as it is simple to implement and does not require any prior knowledge of the environment. In this work we discuss why it is challenging to design an exploration strategy for reinforcement learning agents, discuss some important properties for exploration, highlight some shortcomings of existing methods, and finally propose an exploration strategy which is simple to implement and can be used in conjunction with any reinforcement learning algorithm.

A typical approach to incorporate exploration into a value-based RL agent is to obtain optimistic value estimates. The agent takes greedy actions according to this optimistic value estimate, leading it to take actions that look good either because they have high uncertainty or because the action is actually high value. This approach has been well-developed for the contextual bandit setting, with a variety of algorithms and theoretical results on optimality (Li et al., 2010; Abbasi-Yadkori et al., 2011). Understanding is growing about how to soundly extend these ideas to reinforcement learning, though the theoretical results on estimating and using optimistic values are limited to the linear function approximation setting (Grande et al., 2014; Osband et al., 2016a; Abbasi-Yadkori et al., 2019; Wang et al., 2019).

Though the theory is difficult to extend, there has been a concerted effort to develop and empirically evaluate such optimistic value estimation approaches for the deep RL setting. Bootstrap DQN with priors, for example, maintains an ensemble of action-values, which reflect uncertainty in the value estimates (Osband et al., 2018; 2019). It takes a Thompson sampling approach—which can be seen as optimistic—by sampling one action-value in the ensemble and following it for an entire episode. Another common approach to obtain optimistic value estimates employs the usage of *reward bonuses* (Bellemare et al., 2016; Ostrovski et al., 2017; Burda et al., 2019; Ash et al., 2022). A reward bonus, reflecting uncertainty with respect to the transition, is added to the reward,

increasing the estimated value proportionally for the corresponding states and action.

Most works, however, eschew these directed exploration approaches in favor of simpler, undirected exploration approaches like ϵ-greedy. One potential reason for this is that reward bonus approaches do not encourage *first-visit optimism*. They encourage revisiting a state, if the reward bonus was high in that state; namely, they retroactively reason about uncertainty of states they have seen. The reward bonus cannot encourage visiting a state for the first time. Bootstrap DQN with priors (BDQN), on the other hand, tries to solves this issue by using an ensemble and fixed additive priors for each value function in the ensemble to provide first-visit optimism. Unlike reward bonuses, though, BDQN is more onerous to use. It requires completely changing the algorithm to one that maintains and updates an ensemble, and making key choices like how often to follow one of the value functions in the ensemble before switching. Recent work suggests it is key to have a large ensemble for BDQN (Janz et al., 2019; Osband et al., 2023). Epinets (Osband et al., 2023) can match the performance of BDQN with much less compute, but are arguably even more onerous to implement than BDQN. Our goal is to develop an easy-to-use exploration approach for deep RL, that can easily be added to an existing algorithm, making it less onerous to displace the default ϵ-greedy approach.

To do so, we explore how to directly estimate a *value bonus*. The agent acts greedily according to the value estimate plus this separate value bonus b, namely $\text{argmax}_a\, q(s,a) + b(s,a)$. The value bonus should ideally represent the uncertainty for that state and action. Though this may be the first time this term is used,[1] there are some works that estimate value bonuses. One simple approach is to separate out the reward bonuses and propagate them to learn a second value function, as was proposed for RND (Burda et al., 2019) and later adopted by ACB (Ash et al., 2022). This approach, however, still suffers from the fact that reward bonuses are only retroactive, and the resulting b is unlikely to be high for unvisited states and actions. For the contextual bandit setting, the ACB algorithm actually directly estimates the reward bonus using the maximum over an ensemble of functions, which is high for unvisited actions; but the extension to RL with reward bonuses loses this first-visit optimism. UCLS (Kumaraswamy et al., 2018) and UBE (O'Donoghue et al., 2018; Janz et al., 2019) both directly estimate value bonuses, but are limited to linear function approximation. Dora (Choshen et al., 2018) uses value bonuses that are inversely proportional to visitation counts, which is again difficult to extend to the general function approximation setting.

[1] Usually, b would be called a confidence interval, with $q(s,a) + b(s,a)$ an upper confidence bound. However, we do not use that term here, because for the heuristics we use, it is not clear we get a valid upper confidence bound. Instead, it is a bonus added to the value when deciding which action looks promising.

1.1 Contributions

In this work, we introduce a new approach to obtain value bonuses for reinforcement learning, with an algorithm we call Value Bonuses with Ensemble errors (VBE). Similarly to ACB, we use a maximum over an ensemble, but directly use that maximum as the value bonus, rather than indirectly through reward bonuses. The idea is to sample a random action-value function (RQF)—such as a random neural network—and extract the implicit random reward function underlying this RQF target. The RQF predictor in the ensemble is updated using temporal difference learning on this random reward. Because the RQF target is sampled from the same function class as the RQF predictor, the error can eventually reduce to zero, allowing the value bonus to shrink to zero. These value bonuses are learned separately from the main action-values, and so can be layered on top of many algorithms. In our experiments, for example, we simply use Double DQN (Van Hasselt et al., 2016), and modify the step where the agent selects an action from ϵ-greedy to instead taking the greedy action in the value estimate plus the value bonus. We show that this simple approach is an effective, and scalable method for exploration that improves sample efficiency of learning in a range of domains: from hard exploration gridworlds, to image-based Atari domains.

1.2 More on Related Work

Many algorithms estimate optimistic values via upper confidence bounds in the tabular setting. These include Interval Estimation Q-learning (Meuleau & Bourgine, 1999), E^3 (Kearns & Singh, 2002), R-max (Brafman & Tennenholtz, 2003), UCRL (Auer & Ortner, 2006), UCRL2 (Jaksch et al., 2010), REGAL (Bartlett & Tewari, 2012), Delayed Q-learning (Strehl et al., 2006) and MBIE-EB (Strehl & Littman, 2008). Some of these ideas have been extended to the linear function-approximation setting, maintaining some theoretical guarantees, by RLSVI (Osband et al., 2016a), DGPQ (Grande et al., 2014), Exploration-enhanced POLITEX (Abbasi-Yadkori et al., 2019) and LSVI-UCB (Wang et al., 2019).

Though some theoretically-sound works listed above use a model-based approach, Neu & Pike-Burke (2020) showed that value optimism and model optimism are equivalent. This insight is useful, because value optimism should be simpler to obtain; it supports the approach taken in this work. Closely related, Ciosek et al. (2020) show that fitting random prior functions serve as a computationally tractable approach towards estimating uncertainty in the supervised learning setting, which also motivates the approach taken here.

Chapter 2

Background

Reinforcement Learning (RL) is a paradigm of Artificial Intelligence (AI), where an agent learns to interact with an environment to maximize a scalar reward signal. The agent learns to select actions that maximize the expected sum of future rewards (Sutton & Barto, 2018) or the average reward (Naik et al., 2019; Wan et al., 2021). The agent does not have access to a supervisor, nor does it have access to a dataset of pre-collected examples. Instead, the agent learns by trial and error, receiving feedback in the form of rewards and penalties. In the tabula-rasa setting the agent has no prior knowledge of the environment or about itself, and must learn by interacting with the environment. After each interaction, the agent receives a reward and observes the next state of the environment. The agent then updates it's beliefs about the environment, and uses these beliefs to update it's behavior accordingly. The goal of the agent is thus to learn a policy that maximizes the expected total reward.

This is different from supervised learning, where the agent is given a dataset of examples and a supervisor that provides the correct output for each example. The goal there is to learn a function that maps inputs to outputs. RL bears a similarity with unsupervised learning in that there is no need for labeled data for unsupervised learning algorithms, but the goal however is different. In unsupervised learning, the goal is to learn a representation of the data, or pattern recognition. The key thing that distinguishes RL from other machine learning paradigms is that in RL the agent learns by interacting with the environment, and that an RL agent has autonomy to interact with the environment, cause changes in the environment, learn from these changes, and to ultimately learn to control or maximize the outcome of the interactions.

Formally this agent-environment interaction is modelled as a Markov Decision Process (MDP). An MDP consists of $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$ where $\mathcal{S}$ is the set of states; $\mathcal{A}$ is the set of actions; $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, \infty)$ provides the transition probabilities; $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function; and $\gamma : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition-based discount function which enables either continuing or

episodic problems to be specified (White, 2017). On each step, the agent selects action A_t in state S_t, and transitions to S_{t+1}, according to P, receiving reward $R_{t+1} \stackrel{\text{def}}{=} r(S_t, A_t, S_{t+1})$ and discount $\gamma_{t+1} \stackrel{\text{def}}{=} \gamma(S_t, A_t, S_{t+1})$.

There are many ways for an agent to learn how to interact with the environment and learn a policy that maximizes the expected total reward. One such way to do this is through an iterative process called Generalized Policy Iteration (GPI) (Sutton & Barto, 2018). In GPI, an agent iteratively evaluates the current policy and then improves it. Policy evaluation involves learning an action-value function for the current policy. An action-value function $q^\pi(s, a)$ is the expected total reward (or return) that the agent will receive by following the policy π, after taking the action a in state s. The total reward or the return can simply be defined as the discounted sum of future rewards, or as the average reward (Naik et al., 2019; Wan et al., 2021). The policy is improved by making it greedy with respect to the improved and updated action-value function. The process is repeated until convergence.

In this work, we focus on the value-based approach to RL, and on maximizing the expected sum of future rewards. However, the work we present later in Chapter 3 of this thesis is in no way limited to this setting and can easily be extended to many different RL frameworks. In value-based RL, the agent learns and behaves according to an action-value function $q^\pi(s, a)$. The action-value function can be estimated using Monte Carlo (MC) methods, Dynamic Programming (DP), or Temporal Difference (TD) learning.

MC methods estimate the value function by averaging the returns observed after visiting a state. DP estimates the value function by iteratively applying the Bellman equation, which is a recursive equation that expresses the value of a state in terms of the value of it's successor states. This however requires access to the underlying models of the MDP, which makes it hard to scale when the MDP is large and not known. TD learning estimates the value function by bootstrapping, i.e. by using the value of the next state to estimate the value of the current state. TD learning and MC methods make use of the collected data and do not require access to the underlying models of the MDP, making them more scalable. TD learning is more efficient than MC methods, because it uses bootstrapping, and thus requires less samples to converge. Thus TD learning combines the best of both worlds, it is more efficient than MC methods as it uses bootstrapping, and it does not require access to the underlying models of the MDP.

For a policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, \infty]$, the value for taking action a in state s is the expected discounted sum of future rewards, with actions selected according to π in the future,

$$q^\pi(s, a) = \mathbb{E}_\pi \left[R_{t+1} + \gamma_{t+1} q^\pi(S_{t+1}, A_{t+1}) \middle| S_t = s, A_t = a \right]$$

where $\mathbb{E}_\pi$ means that actions are selected according to π in the expectation. The policy π can be

progressively improved by making it greedy in $q^\pi(s, a)$, then updating the action-values for the new policy, then repeating until convergence (Sutton & Barto, 2018).

In practice, these steps are approximated. The action-values q^π are approximated using q_w parameterized by $w \in \mathcal{W} \subset \mathbb{R}^d$. One algorithm to estimate q_w is Double DQN (DDQN) (Van Hasselt et al., 2016). DDQN is an off-policy algorithm, meaning that it uses a different behavior policy π_b to select actions from the policy it evaluates, which is greedy in q_w. This algorithm uses a target network $q_{\tilde{w}}$ for bootstrapping, giving the following update for one transition (s, a, r, s', γ):

$$w \leftarrow w + \eta\delta\nabla q_w(s, a) \quad \text{for } \delta \stackrel{\text{def}}{=} r + \gamma q_{\tilde{w}}(s', \operatorname*{argmax}_{a'} q_w(s', a')) - q_w(s, a) \tag{2.1}$$

The behavior policy is typically defined to be ϵ-greedy in q_w, but can be any policy that promotes exploration. In this work, we consider an alternative choice for the behavior policy: one that uses a value bonus b, $\pi_b(s) = \operatorname{argmax}_a q_w(s, a) + b(s, a)$. The value bonus should reflect uncertainty in the action-value estimate, encouraging the behavior policy to take an action in a state if it has high uncertainty. It might have high uncertainty if (s, a) is quite different from what it has seen before—meaning it has never been visited—or because the agent has not yet visited it sufficiently often to be certain about it's value. The focus of this work is a new approach for obtaining b for the deep RL setting.

2.1 Exploration background

There are many methods and algorithms that exist for exploration in RL, and there are many categories in which these exploration methods can be divided into, but an important distinction is whether an exploration strategy provides directed or undirected exploration. Undirected exploration works by injecting randomness in the agent's behavior policy which allows the agent to take non-greedy actions at random. Directed exploration allows the agent to take actions that reveal more information about the environment. Although it makes more sense to use directed exploration methods, however in practice undirected exploration methods, like ϵ-greedy or softmax policies are more common. The reason for this is that undirected exploration methods are easier to implement without much overhead, whereas, it is hard to design scalable directed exploration strategies. In this section we will discuss why directed exploration is important, discuss some exploration strategies, some of the shortcomings of existing methods, and how they relate to our work.

2.1.1 Undirected Exploration

Undirected exploration methods are the most common methods used in practice. The most common undirected exploration methods are ϵ-greedy and softmax policies. In ϵ-greedy exploration, the agent selects a random action with probability ϵ, and selects the greedy action with probability $1 - \epsilon$. This allows the agent to take non-greedy actions at some timesteps. This simple method works well in practice for simpler environments where the state space is relatively small. However, since the action selection is an independent random process, the actions taken by the agent do not seek to reveal useful information about the environment. Also, because of random selection the agent can very well select actions that have been taken before or the greedy action itself. This is a very naive way to explore and is very sample inefficient. In practice the agent usually starts with a high value of ϵ and then gradually decreases it over time. This is because the agent needs to explore more in the beginning and then exploit later on. However, this is not a very good strategy since the agent can not know when and how much it needs to explore and exploit. In softmax exploration, also known as Boltzmann exploration, the agent selects an action according to the softmax distribution, which is a distribution proportional to the magnitude of the action-values, parameterized by a temperature parameter τ. The temperature parameter controls the entropy of the distribution, and thus controls the degree of exploration. The stochastic nature of sampling from the distribution allows the agent to take actions that have not yet been tried enough, at random. However, the agent can still take the greedy action with a non-zero probability, and thus the agent can still take actions that have been taken before. Also, it is not clear how to set the temperature parameter τ, as it depends on the nature of the environment which is not known a prior. Another popular undirected exploration strategy for Deep RL is the use of Noisy Networks (Plappert et al., 2017; Fortunato et al., 2018), which in addition to the network parameters, maintains and updates a set of noise parameters which are added to the original network parameters, and are randomly scaled after each optimization step. All these exploration strategies provide some level of random exploration, however, these strategies do not seek to take actions that reveal more information about the environment, and thus are not suitable for complex environments with large state spaces, and are sample inefficient. Undirected random exploration strategies can take exponential times to explore the state space when the goal is at a hard to reach place (Osband et al., 2019).

2.1.2 Directed Exploration

In directed exploration the agent seeks to take actions that reveal more information about the environment. The basic idea of most directed exploration algorithms comes from the principle of Optimism in the Face of Uncertainty (OFU) (Kaelbling et al., 1996; Szepesvári, 2022). The general idea is that whenever the agent is uncertain about an action in a state, the agent should take that action. There can be many sources of uncertainty, it could be that the agent has not taken that

action before or enough times compared to the other actions, or that there is some stochasticity associated with that state either in the reward or the transition function, thus requiring more visits to be certain about the learned estimates associated with that state.

One of the fundamental directed exploration algorithm comes from the bandit literature, and is known as Upper Confidence Bound (UCB) (Agrawal, 1995; Auer et al., 2002; Auer, 2002; Sutton & Barto, 2018; Lattimore & Szepesvári, 2020). In the bandit setting there are K arms, and the agent has to select an arm to pull at each timestep. The agent receives a reward after pulling an arm, and the goal is to maximize the total reward. The agent does not know the reward distribution of each arm $\theta_t(a)$, where $a \in [1, 2, ..., K]$, and thus has to learn it by pulling the arms. In the simplest version of UCB the agent maintains counts of the number of times each arm has been pulled $N_t(a)$, and the estimate of reward function of each arm $\hat{\theta}_t(a)$. The agent then selects the arm based on the reward function estimates and the bonus term which is inversely proportional to the counts associated with that arm: $A_t = \text{argmax}_a \, \hat{\theta}_t(a) + \beta\sqrt{\frac{\log t}{N_t(a)}}$, where β is scaling parameter for the bonus. The agent will select an arm that has been pulled less number of times, and thus is uncertain about it's reward function estimate. This is a very simple algorithm with strong theoretical guarantees and works well in practice. However, it is not clear how to extend this to the multi-step RL setting, where the agent takes actions with respect to a value function, and thus needs a bonus in the value space that can lead the agent to uncertain parts of the state-space later on following a particular policy.

Another way of doing directed exploration is to use Thompson sampling (Thompson, 1933), which has also been extensively studied in the bandit literature (Chapelle & Li, 2011; Agrawal & Goyal, 2012; Gopalan et al., 2013; Agrawal & Goyal, 2013; Russo & Van Roy, 2014). Thompson sampling is based on sampling the reward function estimates $\hat{\theta}_t(a)$ from a posterior distribution $N_t(\mu_{t,a}, \sum_{t,aa})$, where $\mu_{t,a}$ and $\sum_{t,aa}$ is the expected mean and covariance of each arm $a \in [1, 2, ...K]$ updated using Bayesian regression. Thompson sampling provides certain advantages over UCB methods, like not requiring to compute the bonus term entirely, rather Thompson sampling only requires sampling the parameters from the posterior distribution (Russo & Van Roy, 2014). Thompson sampling provides optimism by directly incorporating the covariance of the posterior distribution in the action selection process: $A_t = \text{argmax}_a \, \hat{\theta}_t(a)$. An arm a with higher uncertainty in it's estimates will have a higher variance, which would produce higher values for that arm due to its broad distribution. Thompson sampling also enjoys strong theoretical guarantees. However, this is perhaps even more complex to extend to the RL setting, as a naive implementation would require maintaining a posterior distribution over the action-value function, which can be computationally intractable.

There have been attempts to extend UCB-style (Bellemare et al., 2016; Pathak et al., 2017; Choshen et al., 2018; Burda et al., 2019; Ash et al., 2022), and Thompson sampling (Osband et al.,

2013; 2016a;b; Osband & Van Roy, 2017; Osband et al., 2018; 2019; 2023) exploration methods to the RL setting, which gives us the two main class of directed exploration algorithms that we will discuss here. First is the reward bonus methods, and second is a class of methods that incorporates optimism directly in the action-value estimates.

2.1.3 Reward bonus methods

Reward bonus strategies aim to extend UCB-style exploration methods to the RL setting. The idea is simple, during action selection the agent should pick actions that maximizes the action-value and a bonus term, which corresponds to the uncertainty associated with those actions. However, in the multi-step RL setting the agent has to learn about what will happen in the future and reflect that information in the value estimates. Similarly, the agent at each timestep needs to consider the uncertainty about the future and not just the immediate uncertainty associated with the actions at that step. This makes it challenging to extend UCB-style methods to the RL setting, and requires that the local uncertainty of a state propagates to earlier states (Meuleau & Bourgine, 1999). One way to do this is that the agent in addition to trying to maximize the expected sum of future rewards, should also maximize the expected sum of future intrinsic rewards. The intrinsic reward or the reward bonus can be thought of as a reward representing agent's curiosity or uncertainty. Most simply, the bonus term – proportional to agent's uncertainty – is added to the reward coming from the environment. The agent then estimates the sum of future rewards and the added bonus term, and takes actions that maximizes this bonus incorporated value estimates. Ideally, with more experience the bonus term should decay and the value estimates converge to the true value estimates.

The reward bonus is typically a proxy measure of uncertainty and in literature many different methods have been proposed, ranging from count-based methods (Bellemare et al., 2016; Ostrovski et al., 2017; Martin et al., 2017; Tang et al., 2017; Choshen et al., 2018; Machado et al., 2020) to prediction errors in dynamics function (Stadie et al., 2015; Pathak et al., 2017), and in random auxiliary tasks (Burda et al., 2019; Ash et al., 2022). In general it is hard to keep accurate track of visitation counts with function approximation and large state-spaces. Prediction errors in dynamics function may not be a suitable measure of uncertainty as it can reflect the noise in the environment (Linke et al., 2020; Burda et al., 2019).

Reward bonus strategies, although quite famous, lack an important property for exploration, i.e., they do not provide first-visit optimism. Reward bonus strategies without any optimistic initialization are retroactive, rather than being proactive in terms of providing optimism. An agent has to take an action in a state first for the intrinsic reward to be incorporated in its estimates. This means that the agent can only know about the uncertainty of the state-action pairs it has already visited. This uncertainty is usually referred to as in-sample epistemic uncertainty. However, for

an agent to be able to effectively explore the environment, it needs to be able to take actions that have not yet been tried, and take actions that lead to states where some actions have not been tried. This is referred to as the out-of-sample uncertainty, and the reward bonus methods without additional methods do not incorporate this type of uncertainty.

Another problem with reward bonus strategies is that they can induce non-stationarity in the learning process, even if the environment itself is stationary. The reward bonus, being a measure of uncertainty can initially start off high and then decay with learning and experience, this changes the rewards distribution as it gets added to the original reward. This can sometimes slow down the learning process and requires additional solutions to deal with (McLeod et al., 2021; Ash et al., 2022).

2.1.4 Optimistic value estimates

Instead of estimating local uncertainties and then propagating it to earlier states, another way to do directed exploration is to incorporate optimism directly in the value estimates. The idea is simple, the agent should take actions that maximize the value estimates, and the value estimates should be optimistic. The agent should be optimistic about the value estimates of the states that it has not yet visited (out-of-sample epistemic uncertainty), and the states that it has not yet visited enough times (in-sample epistemic uncertainty). This however is more challenging as it is not easy to get accurate uncertainty estimates for the value function. Kumaraswamy et al. (2018) proposed an algorithm called Upper Confidence Bound Least Square (UCLS), which estimates the uncertainty around value estimates by learning a second value function with the cumulant being the TD error for a state-action pair. The uncertainty value function is combined with the original value function to define an upper bound and the actions are selected from this upper bound. However, this method only provides in-sample epistemic uncertainty estimates, and is restricted to linear-function approximation.

Another class of algorithms called Randomized Least Square Value Iteration (RLSVI) (Wen, 2014; Osband et al., 2016a; 2019) aim to approximate Thompson sampling for RL. The idea is simple, the value function estimates are updated with data perturbed with independent gaussian noise $z \sim N(0, v)$, where v is supposed to represent the variance in the environment. In the linear setting, the RLSVI solution is equivalent to the conjugate Bayesian posterior. The estimates $Q_{\tilde{\theta}_H(s,a)}$ given by RLSVI tend to grow and shrink with the variance of $Q^*_H(s,a)$ due to sampling from the posterior distribution. This is what provides optimism in the value estimates and allows the agent to do deep exploration. However, the original RLSVI algorithm is a fixed-horizon batch algorithm that requires re-using the entire data to update the estimates. Also, the noise $z(s,a)$ has to be sampled once during one iteration or episode. This makes sure that the optimistic value of a state due to the noise $z(s,a)$ is allowed to propagate. Sampling the noise multiple times in

one iteration can cancel out the optimism, as some noise samples would be optimistic and other pessimistic.

The practical online variant of RLSVI uses the discounted RL framework, and relies on ensemble-sampling to estimate the distribution induced by RLSVI. This version maintains an ensemble of value functions and updates them in parallel using distinct datasets for each value function in the ensemble. A short coming of RLSVI is that it requires the noise variance parameter v, which can not be known a prior, as it is supposed to the model the variance of the environment. Another way to incorporate randomization is to use statistical bootstrapping (Tibshirani & Efron, 1993), as is used in Bootstrap-DQN (Osband et al., 2016b). Bootstrap-DQN is an instance of the class of RLSVI algorithms, which provides some advantage over gaussian noise perturbing (Osband et al., 2016b; 2019) like not requiring the variance parameter v. These algorithms incorporate in-sample epistemic uncertainty directly into the value estimates, however, they lack out-of-sample uncertainty incorporation. A simple fix was proposed by Osband et al. (2018), by using fixed additive priors for each value function in the ensemble to provide first-visit optimism.

2.1.5 baseline methods

Here we describe the baseline methods used in this work for comparison on different RL benchmarks. We use algorithms belonging to both the classes of directed exploration methods, i.e., reward bonus methods and optimistic value estimates.

BDQN

The first one is Bootstrap-DQN with additive priors (BDQN) as proposed in Osband et al. (2018). BDQN uses an ensemble of size k action-value functions, which are updated in parallel using distinct replay-buffers for each value function in the ensemble. At the beginning of each episode a value function is uniformly sampled from the ensemble $j \sim \mathcal{U}([1, ..., k])$, and the agent behaves greedy w.r.t Q_j for that episode. It is not clear how and when the value function should be sampled for continuing problems. In Osband et al. (2016b) an ensemble voting policy is used for offline evaluations on Atari benchmark. The statistical bootstrapping accounts for the in-sample epistemic uncertainty, and the out-of-sample epistemic uncertainty is accounted for by using fixed additive priors for each value function in the ensemble.

Let $\gamma \in [0, 1]$ be the discount factor, f_θ be the action-value function parameterized by θ, $f_{\bar{\theta}}$ be the periodically updated target function, p be the fixed additive prior, c be scale parameter for the

Algorithm 1 Bootstrap-DQN with additive priors (BDQN)

Parameters: ensemble size k, bonus scale c, target net update frequency τ, batch size m
Initialize empty buffers $B_{[1...k]} \leftarrow \emptyset$, ensemble $f_{\theta_1}, \ldots, f_{\theta_k}$, target networks $f_{\bar{\theta_1}} \ldots f_{\bar{\theta_k}}$, and fixed additive priors $p_1, \ldots, p_k$ with a standard random initialization
$j \sim \mathcal{U}([1, ..., k])$ // Uniformly sample a value function from the ensemble
Get the initial state s_0
for environment interactions $t = 0, 1, \ldots$ **do**
    Take action $a_t \leftarrow \text{argmax}_{a_t}(f_{\theta_j}(s_t, a_t) + c * p_j(s_t, a_t))$ and observe $r_{t+1}, s_{t+1}, \gamma_{t+1}$
    With probability 0.5 for each buffer add $(s_t, a_t, r_{t+1}, s_{t+1}, \gamma_{t+1})$ to $B_{[1...k]}$
    for i in $[1, \ldots, k]$ **do**
        Sample a mini-batch from B_i
        Update f_{θ_i} using Equation (2.2)
    end for
    if $t + 1 \mod \tau == 0$ **then**
        $f_{\bar{\theta}_{[1...k]}} \leftarrow f_{\theta_{[1...k]}}$ // Updating target networks
    end if
    if $\gamma_{t+1} == 0$ **then**
        $j \sim \mathcal{U}([1, ..., k])$ // Uniformly sample a value function at the end of episode
    end if
end for

prior, and $D = (s_t, a_t, rt, s_{t+1})$ be the data, then the TD update for BDQN is given by:

$$\theta \leftarrow \theta + \eta \delta_\theta \nabla f_\theta(s, a); \quad \delta_\theta \stackrel{\text{def}}{=} \underbrace{r_t + \gamma \max_{a_{t+1}} (f_{\bar{\theta}}(s_{t+1}, a_{t+1}) + c * p(s_{t+1}, a_{t+1}))}_{target} - \underbrace{(f_\theta(s_t, a_t) + c * p(s_t, a_t))}_{prediction} \tag{2.2}$$

In addition to this update, BDQN uses a double-or-nothing bootstrap (Owen & Eckles, 2012), i.e., each transition (s_t, a_t, r_t, s_{t+1}) has a 50% chance of being included in the replay-buffer of each value function in the ensemble. Algorithm 1 shows the complete pseudo-code for BDQN.

DQN-P

DQN-P is a simple modification of BDQN, with only one value function with additive priors, and without any statistical bootstrapping. The reason to add this baseline is to test if the the additive priors can provide the necessary out-of-sample uncertainty and can it be used instead of ϵ-greedy exploration. Algorithm 2 shows the pseudo-code for DQN-P.

Algorithm 2 DQN with additive priors (DQN-P)

Parameters: bonus scale c, target network update frequency τ, batch size m
Initialize empty buffers $B \leftarrow \emptyset$, action-value function f_θ, target network $f_{\bar{\theta}}$, and fixed additive priors p with a standard random initialization
Get the initial state s_0
for environment interactions $t = 0, 1, \ldots$ **do**
    Take action $a_t \leftarrow \text{argmax}_{a_t}(f_\theta(s_t, a_t) + c * p(s_t, a_t))$ and observe $r_{t+1}, s_{t+1}, \gamma_{t+1}$
    Add $(s_t, a_t, r_{t+1}, s_{t+1}, \gamma_{t+1})$ to B
    Sample a mini-batch from B
    Update f_θ using Equation (2.2)
    if $t + 1 \mod \tau == 0$ **then**
        $f_{\bar{\theta}} \leftarrow f_\theta$ // Update the target network
    end if
end for

RND

Random Network Distillation (RND) (Burda et al., 2019) is a reward bonus exploration method that defines the intrinsic reward as a model's prediction error trained to predict random features for observations. Identifying the issues with prediction errors of a model trained to mimic a stochastic transition dynamics model, RND proposes to predict deterministic targets, so that the errors in predictions are indicative of whether the agent has seen that observation enough times during the updates. A model trained to predict a deterministic target for an observation would ideally have less prediction errors for the observations seen many times during the updates, compared to those seen less frequently. This captures the essence of epistemic uncertainty, and allows the prediction error to be used as a reward bonus.

RND uses a randomly initialized network parameterized by $\hat{\theta}$ to define the target function $f_{\hat{\theta}} : \mathcal{O} \to \mathbb{R}^d$, where $\mathcal{O}$ is the observation. A prediction function f_θ is defined that belongs to the same function class as the prediction network $f_{\hat{\theta}}$, meaning both have similar architecture and similar weight initialization methods. The prediction network is then updated using gradient descent to minimize the Mean Squared Error (MSE) $\|f_\theta(x) - f_{\hat{\theta}}(x)\|^2$. The intrinsic reward or the reward bonus is defined as the prediction error of the prediction network $r_{int} = \|f_\theta(x) - f_{\hat{\theta}}(x)\|^2$. RND then uses this intrinsic reward to learn a separate value function to propagate local uncertainties.

The reason for using a separate value function is because making the intrinsic value function non-episodic helps with exploration (Burda et al., 2019; Ash et al., 2022). The idea behind it is that the agent should be curious about a state even if the episode terminates in that state, i.e., the intrinsic return should not be zero for a state because of episode terminating. Since, the value function trained on environment rewards can be episodic – depending on the environment – a combined value function can not be learned on these two different reward streams. Also, it maybe

Algorithm 3 Random Network Distillation (RND) with PPO

Parameters: $N \leftarrow$ number of rollouts, $N_{opt} \leftarrow$ number of optimization steps, $K \leftarrow$ length of rollouts, $t \leftarrow 0$
Initialize $f_{\hat{\theta}}$ target function, f_θ prediction function, and $\pi_{\tilde{\theta}}$ policy parameters with a standard random initialization
for i in $[1, \ldots, N]$ **do**
    for j in $[1, \ldots, K]$ **do**
        take action $a_t \sim \pi(.|s_t)$ and observe $r_{t+1}, s_{t+1}, \gamma_{t+1}$
        calculate intrinsic reward $r_{int_{t+1}} = \|f_\theta(x) - f_{\hat{\theta}}(x)\|^2$
        add $(s_t, a_t, r_{t+1}, r_{int_{t+1}}, s_{t+1}, \gamma_{t+1})$ to optimization batch B_i
        $t \leftarrow t + 1$
    end for
    Compute Return R_i, R_{int_i}, and Advantages A_i, A_{int_i}
    Combine Advantages $A_i^{comb} = A_i + A_{int_i}$
    for k in $[1, \ldots, N_{opt}]$ **do**
        Update policy parameters $\pi_{\tilde{\theta}}$ using PPO on B_i
        Update prediction function f_θ using MSE loss on B_i
    end for
end for

better to learn a separate value functions for each reward stream, as the intrinsic reward function is non-stationary, whereas the environment reward function can be stationary or non-stationary depending on the environment. To incorporate the reward bonus into agent's behavior RND uses a modified version of Proximal Policy Optimization (PPO) (Schulman et al., 2017) that uses the two mentioned value functions for the updates.

Since RND is a reward bonus method, it does not provide the out-of-sample optimism, also since the targets are deterministic it does not provide extra bonus for visiting the stochastic parts of the environments, as they require more visitation. Algorithm 3 provides the pseudo-code for RND.

ACB

Anti-concentrated confidence bonuses (ACB) (Ash et al., 2022) is a reward bonus strategy that tries to extend a UCB-style exploration algorithm to the RL setting. ACB uses an ensemble of linear regressors which is trained to predict independent standard gaussian noise $N(0, 1)$. The reward bonus is defined to be proportional to the maximum deviation over this ensemble from the mean 0. With more updates the ensemble would concentrate around the mean 0, and thus the reward bonus would decay.

Like RND, ACB uses a separate non-episodic value function for the reward bonuses to propagate local uncertainties, and uses PPO to update the policy parameters. To deal with non-stationarity

Algorithm 4 Anti-concentrated confidence bonuses (ACB) with PPO

Parameters: $N \leftarrow$ number of rollouts, $N_{opt} \leftarrow$ number of optimization steps, $K \leftarrow$ length of rollouts, $t \leftarrow 0$
Initialize $w_{\theta_1} \dots w_{\theta_k}$ ensemble of linear regressors, $\pi_{\tilde{\theta}}$ policy parameters, auxiliary policy parameters $\pi_{\tilde{\theta}_{aux}}$ with a standard random initialization
for i in $[1, \dots, N]$ **do**
    for j in $[1, \dots, K]$ **do**
        take action $a_t \sim \pi(.|s_t)$ and observe $r_{t+1}, s_{t+1}, \gamma_{t+1}$
        Compute gradient-features $g_{t+1} \stackrel{\text{def}}{=} g(s_{t+1}, \pi_{\tilde{\theta}}, \pi_{\tilde{\theta}_{aux}})$
        calculate intrinsic reward $r_{int_{t+1}} = \max_l(\langle g_{t+1}, w_{\theta_l} \rangle^2)$
        add $(s_t, a_t, r_{t+1}, r_{int_{t+1}}, s_{t+1}, \gamma_{t+1})$ to optimization batch B_i
        $t \leftarrow t + 1$
    end for
    Compute Return R_i, R_{int_i}, and Advantages A_i, A_{int_i}
    Combine Advantages $A_i^{comb} = A_i + A_{int_i}$
    for k in $[1, \dots, N_{opt}]$ **do**
        Update policy parameters $\pi_{\tilde{\theta}}$ using PPO on B_i
        For all $l \in [1 \dots k]$ sample random targets $y_{[1 \dots k]} \sim N(0, 1)$
        Update ensemble $w_{\theta_{[1 \dots k]}}$ using MSE loss on $y_{[1 \dots k]}$ and B_i
        Update auxillary network parameters $\pi_{\tilde{\theta}_{aux}} \leftarrow \alpha \pi_{\tilde{\theta}} + (1 - \alpha) \pi_{\tilde{\theta}_{aux}}$
    end for
end for

ACB uses gradient-features computed w.r.t a tail-averaged set of policy weights instead of the most recent policy weights being used to interact with the environment. The gradient-features is a representation learning technique that comes from the active learning domain (Sener & Savarese, 2017), and is used instead of the regular last-layer neural network features as input to the linear regressors. In addition, the reward bonuses are computed using a Polyak-averaged version of the ensemble rather than the ensemble itself, and the weights of the ensemble are aggressively regularized towards the initial weights of the ensemble. This is done to prevent the reward bonuses from decaying too quickly.

Like RND, ACB also does not provide out-of-sample optimism. Since the targets are stochastic, it may take more time and visits to learn the targets and thus bonus maybe valid for a longer time. This can be useful if the environment is stochastic, however, the variance in the targets and the variance in the environment may not be the same. Stochastic targets can cause more flailing, even in the parts where the environment is deterministic. Also, for the stochasticity in the targets to model the stochasticity in the environment, it maybe important to sample the target noise once per interaction like in Osband et al. (2019), rather than sampling it before every update when using a replay-buffer. Algorithm 4 provides the pseudo-code for ACB.

Chapter 3

Value Bonuses with Ensemble Errors

In this section, we present VBE, an alternate approach for extending UCB-style exploration to RL by directly estimating the *value bonus*. We first motivate why we need to estimate the value bonus in the RL setting. Then we discuss how to estimate the value bonuses for reinforcement learning, using a novel idea around random target value functions. Finally we put it all together into an optimistic exploration algorithm called Value Bonuses with Ensemble errors (VBE).

UCB-style exploration methods originate from the multi-armed bandit setting, where an agent has k-arms, each with its own reward distribution. The agent must choose which arm to pull at each time step, and the goal is to maximize the total reward. The agent does not know the reward distributions, but can learn them by pulling the arms. The agent maintains estimates of mean reward for each arm, and acts according to the estimated mean rewards. In order for the agent to explore different arms, it needs a bonus on top of the estimated mean rewards, that encourages the agent to pull arms it is uncertain about. This is primarily done by keeping counts for each arm pull and defining a bonus inversely proportional to these counts. The agent then acts greedily with respect to the estimated mean reward plus the reward bonus. In contrast, an RL agent maintains estimates of the value of each state-action pair, and acts according to these value estimates. Thus, we need a bonus on top of the estimated values, or value bonus, that encourages the agent to visit states and take actions it is uncertain about.

To understand why, consider the case where we use prediction errors over an ensemble to define an exploration bonus, as is done in ACB and RND. For an ensemble of size k, we can generate randomly initialized neural networks $f_1, \ldots, f_k$ and update the learned functions $\hat{f}_1, \ldots, \hat{f}_k$ in the ensemble using a squared error: for each (s, a), update each $\hat{f}_i$ using loss $(f_i(s, a) - \hat{f}_i(s, a))^2$. The bonus for any (s, a) can be set to

$$b(s, a) \doteq \max_{i \in [k]} |f_i(s, a) - \hat{f}_i(s, a)| \tag{3.1}$$

Ciosek et al. (2020) show that fitting random prior functions serve as a computationally tractable approach towards estimating uncertainty in the supervised learning setting. Unfortunately, in the reinforcement learning setting, using this bonus directly for action selection will not do what has been called deep exploration (Osband et al., 2019), as this bonus is a local uncertainty estimate, and is likely to concentrate too quickly. We want the agent to reason not just about uncertainty for this state and action, but also about the uncertainty of the states that it leads into.[1]

Instead we want a bonus that estimates uncertainty in the value estimates, or value bonus. Existing methods that aim to extend UCB-style exploration to RL first estimate local uncertainties associated with states using counts, prediction errors, etc., and then propagate these local uncertainties via value learning to get a value bonus. In this work we propose to use prediction errors over an ensemble that directly gives us the value bonus and promotes deep exploration. More specifically, we want to generate a reward function r_i for each f_{w_i}, where the f_{w_i} are updated using standard temporal difference learning approaches. We want the learning dynamics for these value functions to resemble the primary value function, so that they learn at a similar timescale and are more likely to converge to zero once the primary value function has also converged.

In order to measure errors in the value estimates that can be used to define the value bonus, one needs access to the true value function, which is not known a prior. The value of a state depends on the reward function and the policy, i.e., the value of a state can be different for different policies. Thus, we need to define reward and target value functions that are consistent with each other and allows us to easily measure the errors. Consider if we again do the simplest thing: generate a random neural network r_i for each f_{w_i}. Let us assume for now that we have a fixed policy, π. First, it is not clear how we would actually measure the error since we do not know the true value function f_i, namely the expected return using r_i under policy π. Further, this true value function may not be representable by f_{w_i}.

Instead, our proposed approach is to generate a random action-value function (RQF) f_i, and then define rewards consistent with that f_i. Define the stochastic ensemble reward from (S_t, A_t) to be

$$R_{i,t+1} \stackrel{\text{def}}{=} f_i(S_t, A_t) - \gamma_{t+1} f_i(S_{t+1}, A_{t+1}), \tag{3.2}$$

where $A_{t+1} \sim \pi(\cdot|S_{t+1})$ and $\gamma_{t+1} \stackrel{\text{def}}{=} \gamma(S_t, A_t, S_{t+1})$ is defined by the environment. Further, by definition, the action-values of the random prediction function is:

$$q_i^\pi(s,a) \stackrel{\text{def}}{=} \mathbb{E}_\pi \left[R_{i,t+1} + \gamma_{t+1} q_i^\pi(S_{t+1}, A_{t+1}) \middle| S_t = s, A_t = a \right]. \tag{3.3}$$

We show in the following proposition that $q_i^\pi = f_i$.

[1]Note that RND did not use these errors directly for exploration. Instead, they used them as reward bonuses, which can retroactively promote deep exploration, barring the issue that they do not promote first-visit optimism.

Proposition 1. *For all $i \in [k]$, we have $q_i^\pi = f_i$.*

Proof.

$$
\begin{aligned}
q_i^\pi(s,a) &= \mathbb{E}_\pi\left[R_{i,t+1} + \gamma_{t+1} q_i^\pi(S_{t+1}, A_{t+1}) \middle| S_t = s, A_t = a\right] \\
&= \mathbb{E}_\pi\left[R_{i,t+1} + \gamma_{t+1} R_{i,t+2} + \gamma_{t+1}\gamma_{t+2} q_i^\pi(S_{t+2}, A_{t+2}) \middle| S_t = s, A_t = a\right] \\
&= \mathbb{E}_\pi\Big[[f_i(s,a) - \gamma_{t+1} f_i(S_{t+1}, A_{t+1})] + \gamma_{t+1}[f_i(S_{t+1}, A_{t+1}) - \gamma_{t+2} f_i(S_{t+2}, A_{t+2})] \\
&\qquad + \gamma_{t+1}\gamma_{t+2} q_i^\pi(S_{t+2}, A_{t+2}) \Big| S_t = s, A_t = a\Big] \\
&= \mathbb{E}_\pi\Big[[f_i(s,a) \underbrace{-\gamma_{t+1} f_i(S_{t+1}, A_{t+1})}_{\text{cancels}}] \underbrace{+\gamma_{t+1} f_i(S_{t+1}, A_{t+1})}_{\text{cancels}} \\
&\qquad - \gamma_{t+1}\gamma_{t+2} f_i(S_{t+2}, A_{t+2})] + \gamma_{t+1}\gamma_{t+2} q_i^\pi(S_{t+2}, A_{t+2}) \Big| S_t = s, A_t = a\Big]
\end{aligned}
$$

We can keep unrolling this, and these terms will continue to telescope, leaving only the first term $f_i(s,a)$, completing the proof. □

Therefore, updating f_{w_i} with rewards r_i should converge to q_i^π—and so to f_i—because f_i is in the function class of f_{w_i}. This convergence ensures the value bonuses go to zero, which is desired if we want the agent to stop exploring and converge to the greedy policy. Even with a fixed policy, however, this convergence will only occur under certain conditions. Primarily, the failure would be that f_{w_i} gets stuck in a local minima or even that it diverges, due to know issues with temporal difference (TD) learning algorithms combined with neural networks and with off-policy updates.

There is fortunately a large (and growing) literature understanding the convergence behavior of TD algorithms. Under linear function approximation, we know least-squares TD converges at a rate of $1/\sqrt{T}$ to the global solution, even under off-policy sampling (Tagorti & Scherrer, 2015). With the advent of theory for overparameterized networks, TD with a particular neural network function class has been shown to converge to the global solution, under on-policy sampling (Cai et al., 2019). In general, we know that a class of modified TD algorithms, called gradient TD methods, converge even under off-policy sampling and nonlinear function approximation (Dai et al., 2017; Patterson et al., 2022). Convergence under off-policy sampling is key in our setting, because the behavior policy is optimistic but the target policy may be greedy. We expect that under certain conditions on the neural network it might be possible to say that these gradient TD methods converge to global solutions, though to the best of our knowledge, no such work yet exists. We provide a more complete discussion in Appendix A of how this existing theory on convergence of TD applies to our setting.

To understand how our value bonus method provides optimism by implicitly conserving uncertainty in the value estimates consider this example: Consider the following transition sequence:

(s, a, s', a'), $f_w(s, a)$ is the predictor RQF, $f(s, a)$ is the target RQF, and reward function for the predictor RQF is defined as $r(s, a, s', a') = f(s, a) - \gamma * f(s', a')$ (Equation 3.2). Now suppose that the state-action pair (s', a') has been less frequently visited and thus has high prediction error. This would mean that the error or the $bonus(s', a') = |f_w(s', a') - f(s', a')|$ would be high, but how does this high uncertainty affect $bonus(s, a)$? Notice that we update $f_w(s, a)$ by bootstrapping the prediction value of $f_w(s', a')$ (TD learning update), that is the target for $f_w(s, a)$ is $r(s, a, s', a') + \gamma * f_w(s', a')$. This bootstrap target can only be the "true target" $f(s, a)$, if $\gamma * f_w(s', a') == \gamma * f(s', a')$, that is $f_w(s', a')$ is error-free. This means that prediction error for (s,a) or the bonus(s,a) will only go down to zero when the prediction error for all subsequent state-action pairs is zero. Proposition 1 reflects this property. This essentially allows us to conserve epistemic uncertainty in the value estimates, and do directed deep exploration.

Connection to Reward Shaping: It is interesting to note that this work has close similarities with the reward shaping literature (Ng et al., 1999). The reward function defined for each RQF f_{w_i} in equation 3.2 is a difference of potential functions. This is essentially what allows the predictor RQFs to converge to the target RQFs. Similar to the framework in Ng et al. (1999), we can define the RQF framework using parallel MDPs, i.e., for each RQF in the ensemble we can define a parallel MDP M'_i which is identical to the original MDP $M \stackrel{\text{def}}{=} (\mathcal{S}, \mathcal{A}, P, r, \gamma)$ except that the reward function is defined as $r_i(s, a, s', a') = f_i(s, a) - \gamma * f_i(s', a')$. The true value function for the MDP M'_i and for any policy π is given by f_i. Knowing this true value function allows us to directly measure errors in the value estimates and use it as a bonus to do directed exploration in the original MDP M. Similar to the result in Ng et al. (1999), we can show that the optimal optimistic policy of VBE: $\pi^*_b(s) \leftarrow \text{argmax}_{a \in \mathcal{A}}\, q^*_w(s, a) + c\ b(s, a)$ is identical to the optimal policy of the original MDP M, since the bonus term $b(s, a)$ decays to zero using proposition 1.

Connection to BDQN: Though not obvious at first glance, there is a connection between RQFs and random prior functions in BDQN. In BDQN, the value function is $q_\theta = f_\theta + p$ where p is a random prior function that is not updated and f_θ is the function that is updated. Random priors were developed for stationary state distributions—though then applied to control—so let us consider the update for a fixed policy π. The update uses $a' \sim \pi(\cdot|s')$, giving

$$\begin{aligned} \delta &\stackrel{\text{def}}{=} r + \gamma(f_\theta(s', a') + p(s', a')) - (f_\theta(s, a) + p(s, a)) \\ &\Rightarrow r - \underbrace{(p(s, a) - \gamma p(s', a'))}_{reward-bonus} + \gamma f_\theta(s', a') - f_\theta(s, a) \end{aligned}$$

This is a standard update with reward bonus $p(s, a) - \gamma p(s', a')$, and this bonus is the negation of our reward in Equation (3.2). With a fixed policy, we can separate the value function learning into q^π that estimates the values for the rewards and b^π that estimates the values for the reward bonuses. Namely, f_θ consists of $q^\pi + b^\pi$. As these functions converge, $b^\pi(s, a)$ approaches $-p(s, a)$

using the exact same argument as in Proposition 1, just negating the function p. Consequently, $f_\theta(s,a) + p(s,a) = q^\pi(s,a) + b^\pi(s,a) + p(s,a) = q^\pi(s,a) + (b^\pi(s,a) + p(s,a))$ goes to q^π since $b^\pi(s,a) + p(s,a)$ eventually cancels.

This argument is not how randomized priors are presented, but provides another intuitive interpretation. Further, it highlights a key difference BDQN and VBE: BDQN takes a Thompson sampling approach to induce optimism whereas VBE acts greedily with respect to optimistic value estimates. The ensemble in BDQN aims to approximate the posterior distribution over action-value functions, and the additive priors for each value function in the ensemble provides first-visit optimism. The ensemble in VBE also ensures first-visit optimism, but in a different way compared to BDQN. In BDQN the additive priors in the ensemble must be high value to provide optimism for a state-action pair. Whereas in VBE, the predictor and target RQFs should be different from each other to provide optimism. This can be achieved by defining an ensemble and randomly initializing the RQFs (Ciosek et al., 2020). It can also be noted intuitively that it is easier to define two functions that produce different values through random initialization, than to define a prior function which ensures high values through random initialization. In our experiments we noted that VBE relies on much smaller ensemble sizes – in some experiments even one RQF is enough – compared to BDQN which requires a large ensemble to provide sufficient exploration.

Value Bonuses with Ensemble Errors: We now provide the Value Bonuses with Ensemble Errors (VBE) algorithm in this section. The pseudocode, in Algorithm 5, is for the case where the base algorithm is Double DQN, but it is possible to swap in many different off-policy value-based algorithms. Even actor-critic, which explicitly maintains a critic q_w, could easily incorporate the value bonuses by using instead an optimistic critic. For the purposes of this paper, however, we restrict our focus to Double DQN.

The ensemble value functions are updated on the same target policy as Double DQN, namely the greedy policy in q_w. This choice comes from the fact that we want to understand uncertainty in the values for the target policy. The update is similar to Double DQN, except the actions are sampled according to q_w rather than f_{w_i}, and we use the ensemble reward r_i defined above in Equation (3.2):

$$w_i \leftarrow w_i + \eta\delta_i \nabla f_{w_i}(s,a) \quad \text{for } \delta \stackrel{\text{def}}{=} r_i + \gamma f_{\tilde{w}_i}(s', \operatorname*{argmax}_{a'} q_w(s',a')) - f_{w_i}(s,a) \tag{3.4}$$

On each step, we only update one RQF predictor. Updating the entire ensemble is expensive, and arguably unnecessary. There are multiple ways to control the magnitude of the value bonus, and how quickly it decays. One way is the size of the ensemble, where the larger the ensemble, the more slowly this bonus should decay. Updating each RQF predictor less frequently, however, will also cause the bonus to decay more slowly. It both allows us to make the ensemble smaller, and ensure that regardless of the ensemble size, the computation per-step is simply double that of Double

Algorithm 5 Value Bonuses with Ensemble Errors (VBE)

Parameters: ensemble size k, bonus scale c, target net update frequency τ, batch size m
Initialize empty buffer: $B \leftarrow \emptyset$, action-value function: q_w, target RQFs: $f_i, \dots f_k$, predictor RQFs: $f_{w_1}, \dots, f_{w_k}$, and target networks: $q_{\tilde{w}}$, $f_{\tilde{w}_1}, \dots, f_{\tilde{w}_k}$
Optimistic behavior policy: $\pi_b(s) \leftarrow \text{argmax}_{a \in \mathcal{A}}\, q_w(s,a) + c\; b(s,a)$
where $b(s,a) \leftarrow \max_{i \in [k]} |f_{w_i}(s,a) - f_i(s,a)|$
Get the initial state s_0
for environment interactions $t = 0, 1, \dots$ **do**
    Take action $a \leftarrow \pi(s_t)$ and observe $r_{t+1}, s_{t+1}, \gamma_{t+1}$
    Add $(s_t, a_t, r_{t+1}, s_{t+1}, \gamma_{t+1})$ to the buffer B
    // Update the action-values using the Double DQN update
    Sample a mini-batch and use update in Equation (2.1) to update q_w
    // Update one randomly select ensemble value function
    Sample i from $[k]$ uniform randomly
    Sample a mini-batch from B and update f_{w_i} using Equation (3.4) where
    for each (s, a, r, s', γ) we replace r with $r_i \stackrel{\text{def}}{=} f_i(s,a) - \gamma f_i(s', \text{argmax}_{a' \in \mathcal{A}}\, q_w(s', a'))$
    if $t + 1 \mod \tau == 0$ **then**
        $\tilde{q}_w \leftarrow q_w$ and for all i, $f_{\tilde{w}_i} \leftarrow f_{w_i}$
    end if
end for

DQN: one update to the main value function and one update to an RQF predictor.

We can again ask what happens to our value bonuses in VBE. Ideally, they eventually converge to zero, with the action-values converging and the behavior and target policies both converging to a greedy policy. This scenario goes beyond the convergence conditions discussed above in Section 3 for fixed policies. In VBE, both our behavior policy and target policy are changing with time. Unfortunately, theory around TD does not address this scenario. There are some results for a fixed behavior policy for double Q-learning under linear function approximation (Zhao et al., 2021), or for a variant of DQN with a fixed dataset (Wang & Ueda, 2022). The issue with a changing behavior policy is that it changes the relative importance of states in the objective, and so the best value function may change as it changes how it trades off errors across states. In our realizable setting, this changing importance may be less important, because our RQF predictor can perfectly represent the target. In our own experiments, we found the value bonuses did always converge to zero. Nonetheless, we know of no theory that would allow us to guarantee this.

Chapter 4

Experiments

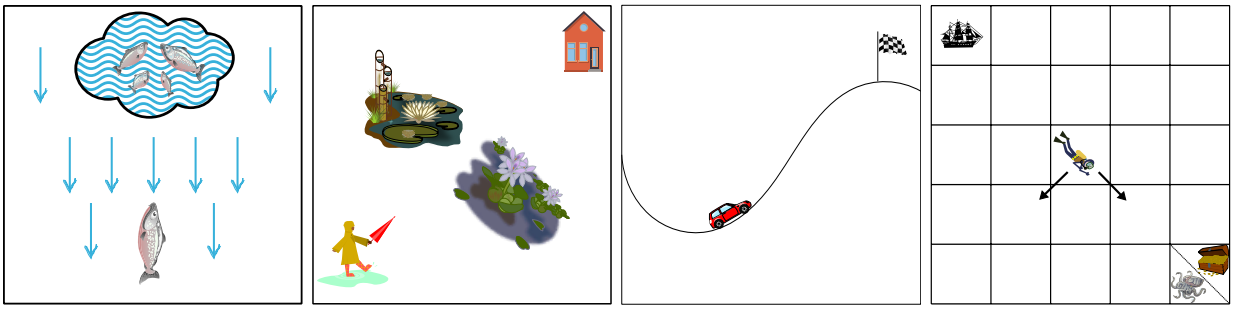
We evaluate our proposed algorithm on four classic exploration environments and six Atari environments, particularly in comparison to BDQN and the reward bonuses approaches ACB and RND. We first investigate the algorithms in a pure exploration setting, on DeepSea, where we evaluate state coverage. Then we compare performance on the classic environments, and investigate the impact of the bonus scale and number of RQFs in the ensemble. We also compare variants of VBE which use ACB and RND's reward bonus strategy to to estimate the value bonuses (VB ACB and VB RND). We conclude with experiments in Atari, particularly highlighting how to scale VBE to this setting.

4.1 Environments

The four classic exploration environments are Sparse Mountain Car, Puddle World, River Swim and DeepSea (Figure 4.1). These four environments have varying requirements for exploration: DeepSea and River Swim are considered hard exploration environments, whereas Puddle World and Mountain Car require less exploration. The full details for these environments are in Appendix A.1, but we list a few key details here.

Mountain Car has two-dimensional continuous inputs, with a sparse reward structure: the agent only receives a reward of 1 at the goal and 0 otherwise. **Puddle World** also has two-dimensional continuous inputs, noisy actions and highly negative rewards in puddles along the way to the goal.

River Swim and **DeepSea** were both designed as hard exploration problems, requiring persistent behavior with likely failure under dithering, ϵ-greedy exploration. River Swim resembles a problem where a fish tries to swim upriver, with high reward (+1) upstream which is difficult



(a) River Swim (b) Puddle World (c) Mountain Car (d) Deepsea

Figure 4.1: A visual depiction of the four classic control environments.

to reach and, a lower but still positive reward (+0.005), which is easily reachable downstream. River Swim is a continuing environment and has a single continuous state dimension in $[0, 1]$, with stochastic displacement when taking actions left or right. One seemingly innocuous but important point for this environment is that we flipped the observation such that the high reward is at observation 0 and the lower reward is at observation 1. We did so because the standard random initialization and ReLU activation often results in a higher value for a higher input, thus favouring the correct action in this case. We found deep RL agents trivially solving this original variant, but not for the reasons we would like! This other variant removes this inadvertent bias without changing the problem structure or difficulty in any way.

DeepSea is similar to River Swim, but is an episodic grid world environment of size $N \times N$. Reaching the high-reward state requires the agent to take the action to go right every time. However, there is a penalty of $\frac{0.01}{N}$ for taking the action right, except for the right most state where the agent gets a reward of 1 for taking the right action. A policy that explores uniform randomly has the probability of 2^{-N} of reaching the goal state in each episode.

4.2 Algorithms and Experimental Settings

The environments uses slightly different evaluation metrics. River Swim is continuing, so we report accumulated reward over learning. For both DeepSea and Puddle World, we report the undiscounted episodic return. For Mountain Car, we report the discounted return, because for every successful episode, the undiscounted return is 1 and so not meaningful in this sparse variant. For all episodic environments, we report steps on the x-axis and the corresponding episodic return on y-axis. For one run, this results in a step function, but averaged across runs we get a measure of average episodic return after t steps. All results in the classic environments use 50000 steps and 30 runs, except for DeepSea which uses 10000 episodes and 5 runs. The default grid size for DeepSea, unless mentioned, is 50, which is the largest grid size we experiment with.

Across problems we compare VBE with DQN with additive priors (DQN-P), BDQN, the released variants of ACB and RND that use PPO [1], and their DDQN-based variants: VB ACB and VB RND. DQN-P simply adds an additive prior to DQN, like BDQN has; it can be seen as BDQN with one value function in the ensemble and without bootstrapping. For both VBE and BDQN, we test using 1, 2, 8 and 20 value functions in the ensembles and bonus scales of 1, 3 and 10. ACB and RND also use the same bonus scales. To match their original implementations ACB uses an ensemble of 128 to estimate the reward bonus, and RND uses two deep neural network with multiple (64) nodes in the final layer as the target and predictor network for the reward bonus. All methods use the same neural network architectures, detailed in Appendix A.1.

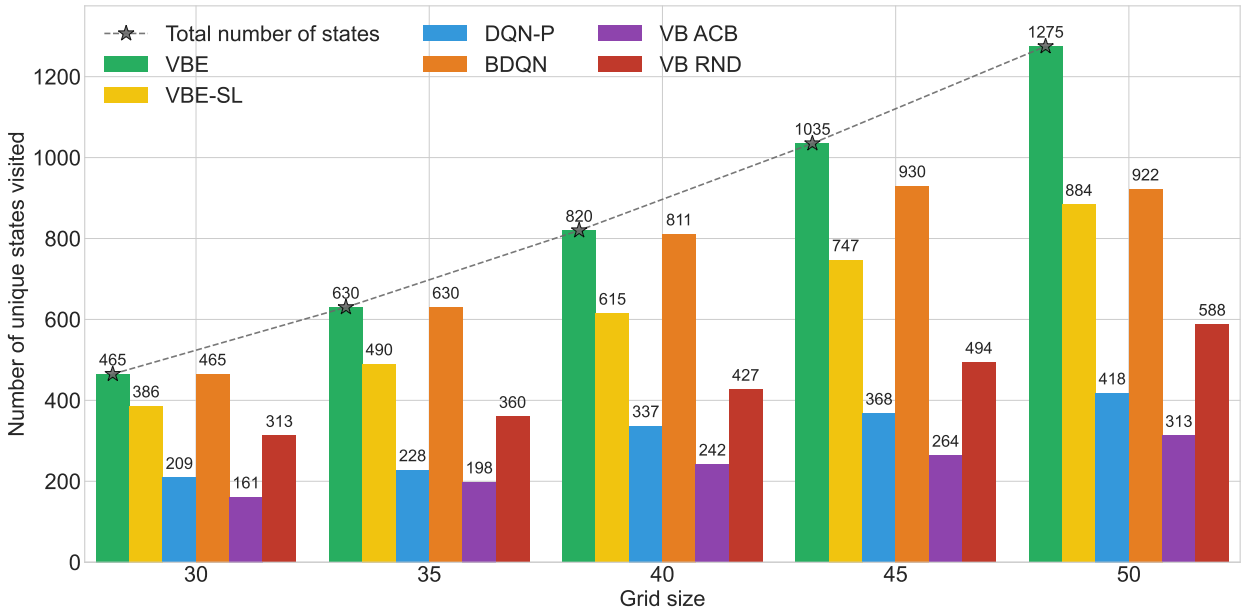
We also include variants of VBE to provide evidence for the way we estimate the value bonuses with our ensemble errors. We include VBE-SL, meaning that instead of the TD update, we use a supervised learning update. We discussed in Section 3 that the errors for VBE-SL are likely to reduce too quickly, resulting in insufficient exploration; we test that hypothesis here. We also evaluate DDQN-based variants of ACB and RND which use the reward bonuses underlying ACB and RND to learn the value bonus and replace them with VBE's ensemble value bonus, i.e., VB ACB and VB RND. As originally proposed by the authors of ACB and RND, we make the reward-bonus value function non-episodic. VB ACB, VB RND and VBE-SL are otherwise exactly the same as VBE, including using DDQN, defining the value bonus using the ensemble error and using the value bonus in the same way.

4.3 Pure exploration

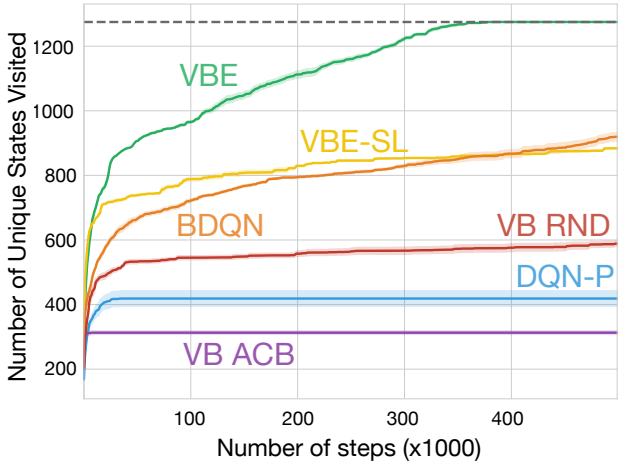
We first test how effectively the agents cover the state-space in Tabular DeepSea environment with increasing grid sizes. For this tabular setting, the agents are otherwise same as the other experiments, except the function approximation is linear on a one-hot encoding. In the pure exploration setting, the agents do not receive any reward from the environment, and the behavior is totally dependent on the exploration strategy employed by the agent. This can be achieved by setting the reward function to be zero for all states and actions. BDQN relies on bootstrapping, ensemble sampling and additive priors for it's exploratory behavior. DQN-P relies on additive priors. VB ACB and VB RND rely on the reward bonuses and the intrinsic value function for exploration. VBE and VBE-SL rely on the ensemble error-based value bonus for exploration.

Figure 4.2a shows that VBE covers the entire state-space for different grid sizes. BDQN is able to cover the state-space for a grid size of 30 and 35, but starts to degrade after that. DQN-P fails to cover the state-space as it only uses one action-value function with additive priors. For the additive priors to provide optimism for different parts of the state-space an ensemble is required as

[1] See https://github.com/JordanAsh/acb/tree/main



(a) State coverage in Deepsea of different grid sizes



(b) Progression of unique states visited (grid size 50)

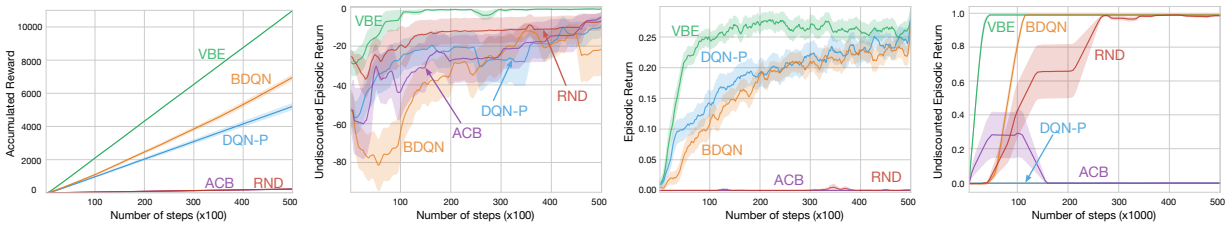
Figure 4.2: Contrasting the state coverage abilities of exploration algorithms in DeepSea. In (a) each bar corresponds to the total number of unique states visited by an agent after completing 10,000 episodes. The black stars indicate the total number of unique states visited for each grid size. Notably, VBE covers the entire state space, even for the larger grid sizes. (b) displays the progression of unique states visited by agents over the course of learning for Deepsea with grid size 50. The dotted line represents the total number of unique states (1275) in this environment. It provides evidence that VBE consistently explores new states at a significantly higher rate.

in BDQN. Both VB ACB and VB RND fail to cover the state space, with VB ACB covering even less than DQN-P. This outcome is not surprising, given that neither approach ensures first-visit optimism. VBE-SL at least includes first-visit optimism, encouraging the agent to take an action in a state if it has not done so before. But, as expected, it does not explore as much as VBE, due to its value bonuses decaying too quickly and not capturing in-sample epistemic uncertainty.

These suboptimal behaviors are emphasized in Figure 4.2b for a grid size of 50. All methods initially start exploring a similar number of states, easily reaching around 300 unique states. VB ACB, DQN-P and VB RND largely stop visiting new states very early in learning, though VB RND

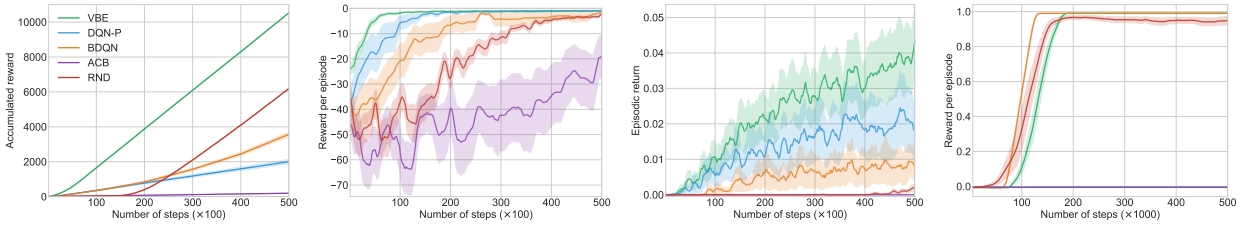
is slowly increasing the number of states it visits. BDQN and VBE-SL across in their behavior, with VBE-SL exploring more early, possibly due to better first-visit optimism. Over time, however, BDQN starts to catch up and then surpasses VBE-SL. One reason why BDQN is slow and fails to explore the state-space of bigger grid sizes in the given time budget maybe due to the random sampling of action-value functions from the ensemble at the start of the episode. Since BDQN uses the sampled action-value function for the entire episode, it can get stuck with a value function that is not optimistic for a region in the state-space. VBE is the only algorithm that maintains a consistent increase until it has seen all states. It is also important to note that VBE is able to cover the state-space for all grid sizes with just one RQF in the ensemble. This result is very significant as other ensemble methods use an ensemble size of 20 – maximum allowed in this experiment – and still fail to cover the state-space. Table A.1 in Appendix A.1 shows the best performing ensemble size k and bonus scale c, for each agent in this experiment.

4.4 Comparison in Classic Environments



(a) River Swim (b) Puddle World (c) Mountain Car (d) Deepsea

Figure 4.3: Comparison of online performance in River Swim, Puddle World, Mountain Car, and Deepsea. In all domains, higher on y-axis is better. The x-axis denotes the number of interaction steps with the environment. The shaded region corresponds to standard errors.



(a) River swim (b) Puddle world (c) Mountain car (d) Deepsea

Figure 4.4: Comparing VBE with baseline agents in four classic control environments, with tile-coded features and a linear-layer.

In this section we compare VBE with DQN-P, BDQN, ACB, and RND. ACB and RND here use PPO as originally proposed. In Figure 4.3, VBE learns faster and reaches the best final performance in all four environments. Surprisingly, DQN-P is competitive with BDQN in three out of the four environments. In DeepSea, where persistent optimism is essential to reach the state with high

reward, DQN-P fails. In Riverswim which is continuing and also has a high rewarding state that can be hard to reach, DQN-P accumulates marginally lesser reward. Since Riverswim is continuing, BDQN can get stuck with a value function from the ensemble that may not be optimistic, requiring the agent to wait for resampling from the ensemble. ACB and RND fail to learn in both the sparse reward domains Riverswim and Mountain Car. In Puddle World and Deepsea which have a denser reward structure, they perform better. RND learns slowly in both, whereas ACB is competitive in Puddle World, but does poorly in Deepsea. In Appendix A.2 we study the impact of the ensemble size and bonus scale on the performance of VBE.

4.4.1 Linear function approximation

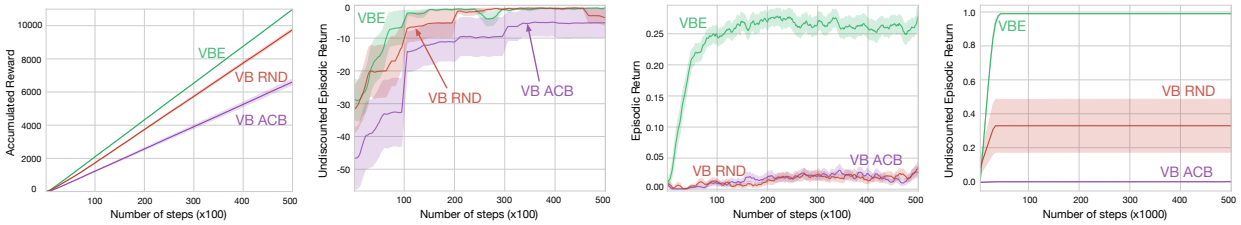
In this section we test VBE and the baseline agents on the same four classic environments with tile-coded features and a single linear-layer network. For Riverswim we use the following tile-coding parameters: $(tiles = 4, tiling = 32, features = 128)$, Puddle world: $(tiles = 5, tiling = 5, features = 128)$, and Mountain car: $(tiles = 4, tiling = 16, features = 512)$. The results in Figure 4.4 are similar to their neural network counterpart results in Figure 4.3. In Riverswim, RND does well and even surpasses BDQN in terms of performance. ACB, however, still fails on Riverswim. In Puddle world RND is comparable towards the end of training, and ACB is much slower. DQN-P outperforms BDQN in Puddle world, and Mountain car, whereas both ACB and RND fail in Mountain car. In Deepsea we see that DQN-P and ACB fail to learn the optimal policy. RND learns relatively quickly but then fails to stick to the optimal policy and thus collects less reward per episode throughout.

4.5 Alternative Choices for the Value Bonus

We compared VBE to VB ACB and VB RND for pure exploration; now we do so for the four classic environments. VB ACB and VB RND are another natural way to estimate value bonuses—albeit missing first-visit optimism—and help validate our new approach to estimating value bonuses.

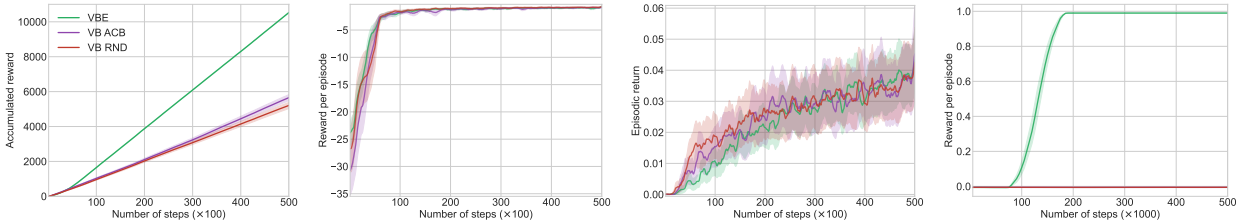
Reward bonus methods can be used and implemented in many different ways. Two of the baseline agents that we use in our experiments, ACB and RND use PPO as their base algorithm, as originally proposed. In this section we test these two reward bonus methods in the value-based setting, and refer to them, respectively, as VB ACB and VB RND. These agents are similar to the ones used in Section 4.3, except here the agent observes the environment reward and updates the value function along with the intrinsic value function. The base algorithm is DDQN, and the behaviour policy is greedy with respect to the learned value estimates plus the scaled intrinsic value function, same as VBE. This experiment is to test and highlight the efficacy of our value bonus strategy compared to reward bonus methods which have known issues like dealing with

non-stationary reward bonuses, and their inability to provide first-visit optimism.



(a) River swim (b) Puddle world (c) Mountain car (d) Deepsea

Figure 4.5: Comparing online performance VBE to two alternative ways to estimate value bonuses, namely estimating a value function on the reward bonuses given by RND and ACB. The shaded regions corresponds to standard errors.



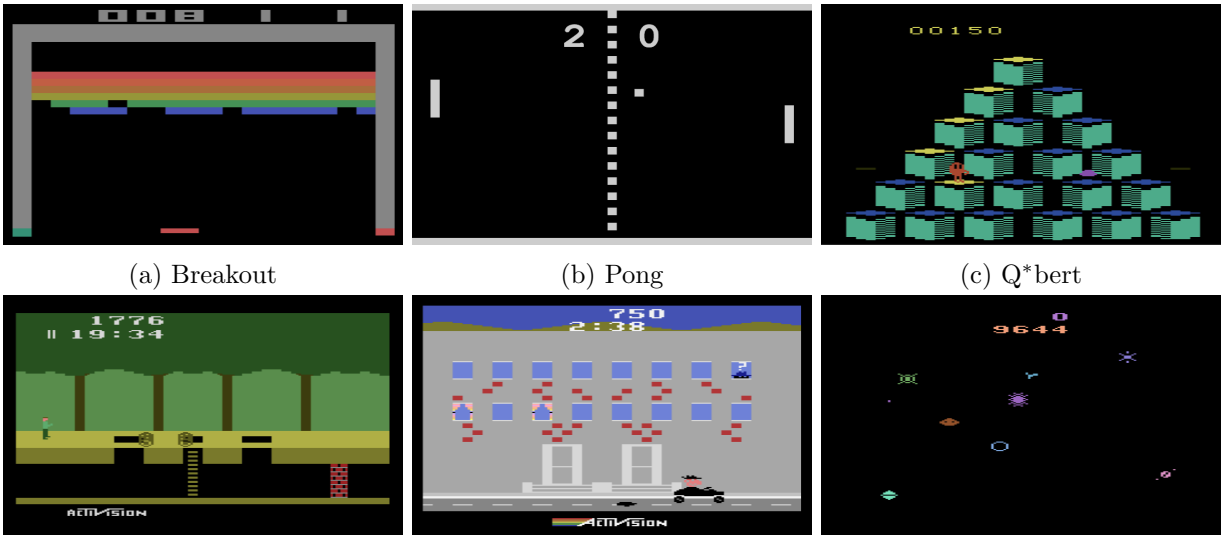
(a) River swim (b) Puddle world (c) Mountain car (d) Deepsea

Figure 4.6: Comparing VBE with VB ACB and VB RND in the linear setting with tile-coded features.

In Figure 4.5 we see that VBE outperforms VB ACB and VB RND in all four environments. Similar to their PPO versions in Figure 4.3, they fail in Mountain Car and are competitive in Puddle World. However, behavior is quite different in Deepsea and Riverswim. VB RND almost matches VBE in Riverswim, and VB ACB has significantly improved compared to its PPO version. But, in Deepsea, they both perform notably more poorly, especially VB RND here fails in most runs but succeeded with its PPO version. The primary difference between them is using DDQN as a base algorithm, rather than PPO. In Riverswim, the VB variants of these reward bonuses is more competitive than the corresponding PPO variants earlier, perhaps due to the off-policy updates used for learning the value function here.

4.5.1 Linear function approximation

Similar to Section 4.4, here we compare VBE with VB ACB and RND in the linear setting with tile-coded features. In Figure 4.6 we see that VBE outperforms VB ACB and VB RND in the two hard exploration environments: Riverswim and DeepSea. In Riverswim VB ACB does better than its PPO counter part in Figure 4.4a. VB ACB and RND perform comparable to VBE in Puddle world and Mountain car. Both VB ACB and RND fail in Deepsea.



(a) Breakout (b) Pong (c) Q*bert

(d) Pitfall (e) Private Eye (f) Gravitar

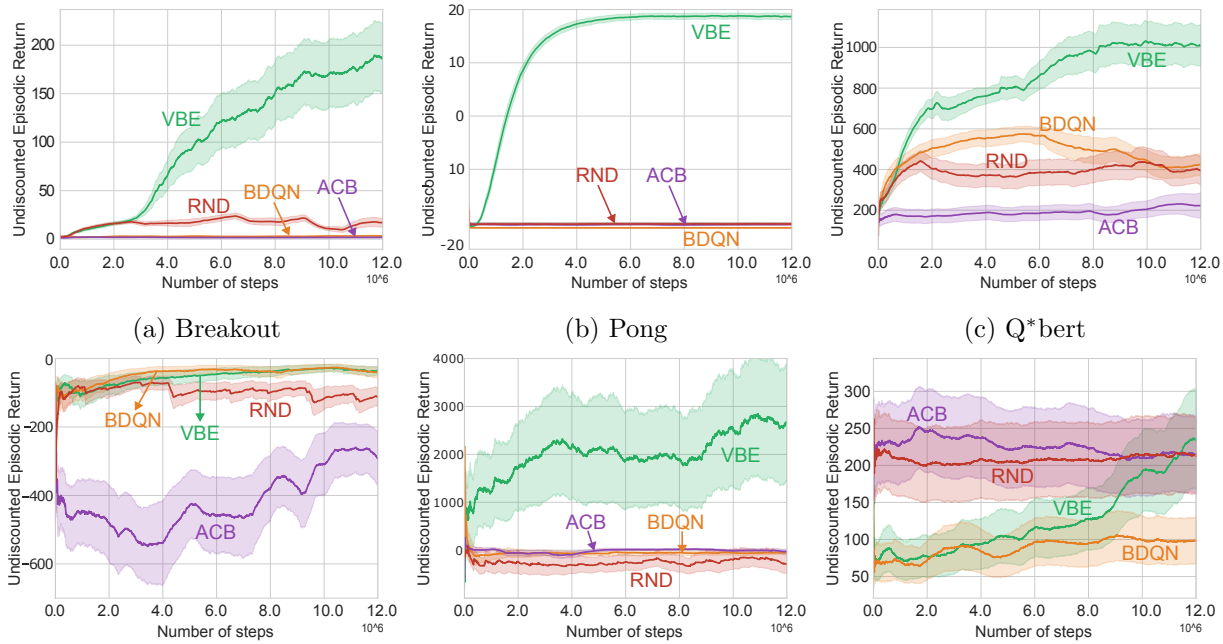
Figure 4.7: A visualization of the Atari environments used in our experiments.

4.6 Atari

In this section we test VBE on several hard exploration Atari games (Figure 4.7), namely Private Eye, Pitfall, Gravitar (Burda et al., 2019), and also on Breakout, Pong and Qbert. As is standard, we combine four consecutive frames to make the observation ($4 \times 84 \times 84$), and update VBE ever four steps. We do 3 runs for all the agents for 12 million steps.

In VBE the target and the predictor RQFs have 3 CNN-layers, followed by 2 non-linear layers (representation network) and a final linear output-layer. We only update the final layer of the predictor RQFs, and initialize the predictors to have the same representation network as the target RQFs, so to have learnable targets. In practice we found that even 1 RQF works well for VBE, so that is what we use in our experiments. We chose this configuration because it was faster to run and seemed to be more stable than updating the whole network. We include the comparison to the variant of VBE where we update the whole network for the RQFs in Section 4.8.

For BDQN we use an ensemble size of $k = 10$. As in the original BDQN implementation, each value function in the ensemble uses a shared representation network. The additive priors have the same network architecture as the value functions. ACB and RND agents do 128-step roll-outs and then do 4 epochs of network updates using PPO. To make sure that each agent gets the same number of interactions with the environment and to match our online setting, we run ACB and RND with one agent interacting with the environment instead of running multiple parallel agents. ACB uses an ensemble size of $k = 128$ for computing the reward bonus, and RND uses a CNN-based target and predictor. Note that VBE runs at least three times faster than BDQN, ACB and RND.



(a) Breakout (b) Pong (c) Q*bert

(d) Pitfall (e) Private Eye (f) Gravitar

Figure 4.8: Comparing online performance in six Atari games, with shaded regions corresponding to standard errors. The environments in the second row are considered to be more challenging from an exploration perspective. The x-axis is the number of environment interaction steps in millions, and the y-axis is the online Undiscounted Episodic Return, for which higher is better.

In Figure 4.8 we see that VBE greatly outperforms the other algorithms in four out of six environments, and is competitive if not better in the remaining two environments. This result is starkly different from the prior work, and is due to the fact that we examine early learning. RND was originally trained on around 2 billion frames, and ACB on around 20 million steps with data coming from 128 parallel agents. In Pitfall all algorithms except ACB are competitive, with VBE and BDQN performing the best. In Gravitar, RND and ACB perform well from the beginning but do not improve overtime. VBE is slow initially but eventually surpasses both ACB and RND. Overall, these results show that VBE scales to more complex deep RL settings and results in sample efficiency improvements in early learning in several Atari environments.

4.7 Atari: Alternative Choices For The Value Bonus

In this section we compare VBE with DDQN-based variants of ACB and RND, denoted as VB ACB and VB RND, on Atari to see early learning. In Figure 4.9 we can see that the VB ACB and VB RND are much more sample-efficient than their PPO-based counterparts. However, VBE still performs better in general. VBE performs the best in four out of six environments, i.e., in Breakout,

Pong, Pitfall, and Privateeye. In Qbert, VBE is competitive with the baselines and performs better than VB ACB. In Gravitar, both VB ACB and VB RND perform better than VBE initially, but VBE surpasses VB ACB towards the end.

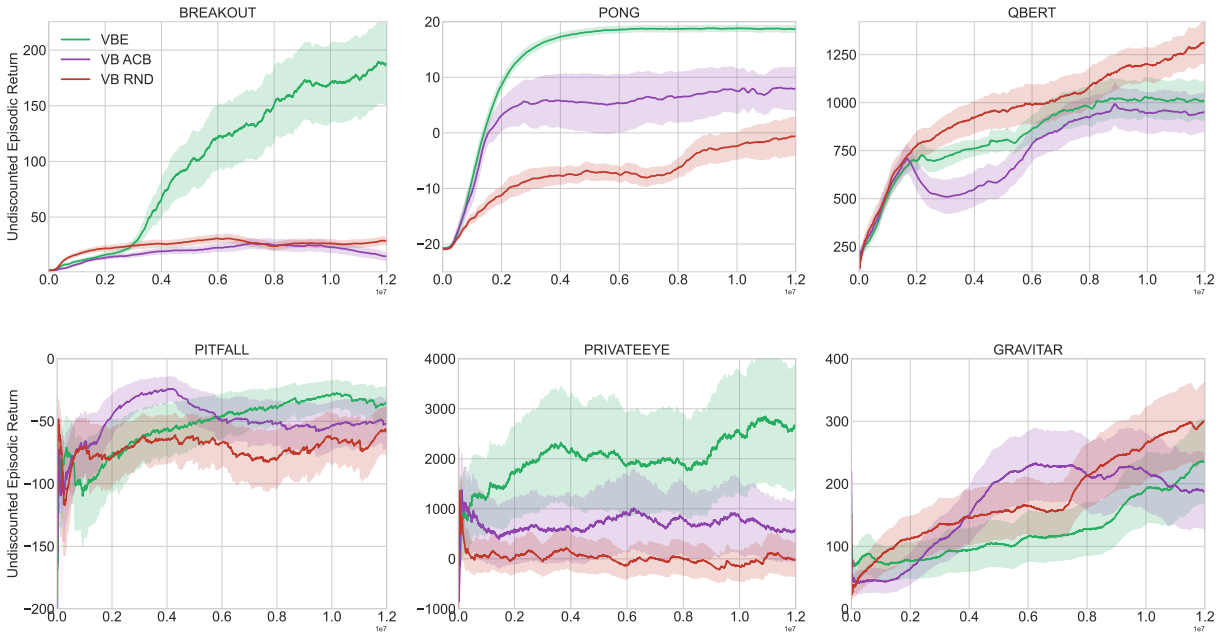


Figure 4.9: Comparing the online performance of VBE, VB ACB and VB RND on six Atari environments on a budget of 12 million steps.

4.8 Atari: An alternate variant of VBE

In this section we compare two variants of VBE on the same six Atari games. VBE as mentioned in Section 4.6 only updates the final output layer of predictor RQFs. We implement a version of VBE, VBE-CNN which updates the complete CNN architecture for predictor RQFs. Since we are updating the complete network the predictor and target RQFs do not need same weights for the representation network. To ensure that the magnitude of errors is not too small, we initialize the target RQFs with a scale of 1 (scale parameter in Orthogonal initialization), and initialize predictor RQFs with a scale of 0.01. Note that this choice of scale was adopted from ACB's public implementation by Ash et al. (2022). We use an ensemble size of $k = 1$ for both these agents, and a bonus scale of $c = 10$ for all games, except for Pitfall for which we use a bonus scale of $c = 1$ for all the agents. In Figure 4.10 we see that both agents continue to learn and improve at around 12 Million frames. VBE is better than VBE-CNN in Breakout, Pong and Privateye. However, in Pitfall VBE-CNN converges to a policy that results in no negative reward at around 6 Million

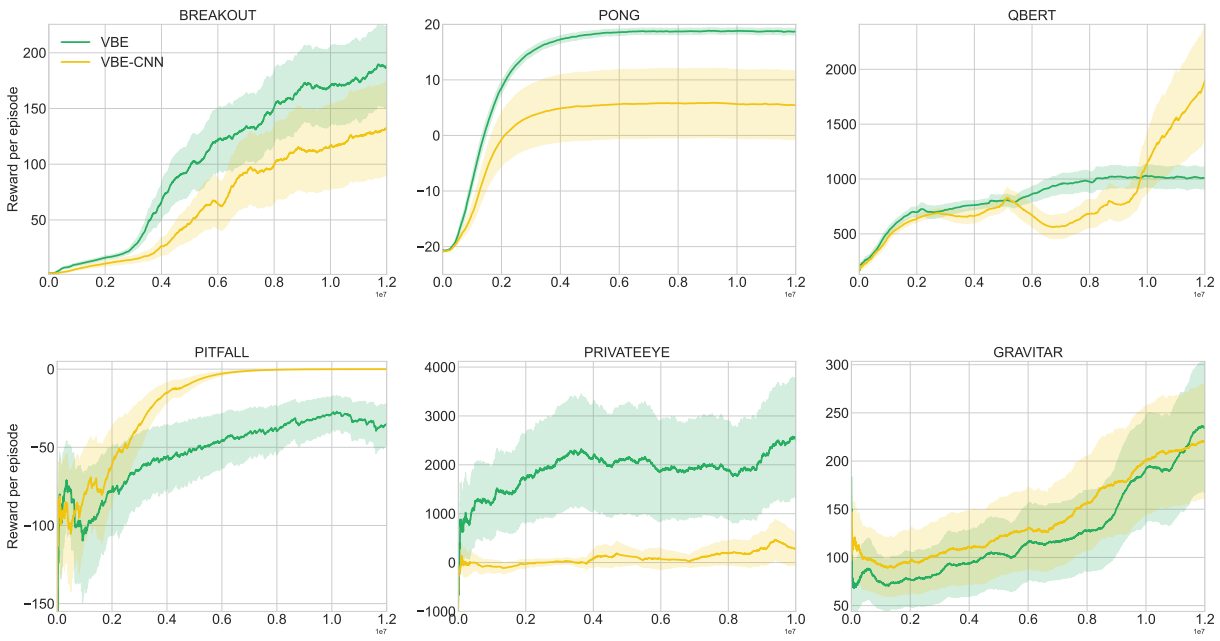


Figure 4.10: Comparing the online performance of VBE and VBE-CNN on six Atari environments. For Private Eye we compare the agents for 10 Million steps, and the rest for 12 Million steps.

frames. This result is quite impressive as Pitfall is a very hard exploration environment and most agents take a lot more data to learn a good policy in this environment. VBE does exceptionally well on Private Eye, which is also a very hard exploration environment. In Q*bert we see that both agents are comparable uptill 6 Million frames after which VBE-CNN first decreases in performance but eventually surpasses VBE towards the end. Finally, in Gravitar we see that both agents are competitive and continue to improve. Both variants of VBE do well in all six Atari environments, and demonstrate sample efficiency by early learning. Although we do not run the agents for as long as the baselines, VBE is still able to learn much quickly and in some environments even outperforms baselines trained on much more data. Table 4.1 shows the mean episodic return observed during the last 500 episodes of online training.

	VBE	VBE-CNN
Breakout	**170.17**	73.08
Pong	**18.81**	5.65
Qbert	963.73	**2026.31**
Pitfall	-18.65	**0.0**
Privateeye	**2416.05**	242.12
Gravitar	**247.76**	214.6

Table 4.1: Mean episodic return observed during the last 500 episodes of online training.

Chapter 5

Conclusion

In this work we introduced a new approach to do directed exploration in deep RL, called Value Bonuses with Ensemble errors (VBE). The utility of value bonuses is that it is simple to layer on top of an existing algorithm: the value bonuses are separately estimated and only impact the behavior policy. Improving how we estimate value bonuses, therefore, provides a promising path to replacing simple, but undirected exploration strategies like ϵ-greedy. To date, the primary way to estimate value bonuses has been to estimate a separate value function on reward bonuses, as was done for ACB and RND. This approach, however, does not encourage first-visit optimism; it only encourages revisiting an action once a reward bonuses was observed. We show that, in general, ACB and RND do not provide effective exploration, in classic environments and several Atari environments, and that VBE consistently outperforms BDQN.

Bibliography

Yasin Abbasi-Yadkori, Dávid Pál, and Csaba Szepesvári. Improved algorithms for linear stochastic bandits. *Advances in neural information processing systems*, 24, 2011.

Yasin Abbasi-Yadkori, Nevena Lazic, Csaba Szepesvari, and Gellert Weisz. Exploration-enhanced politex. *arXiv preprint arXiv:1908.10479*, 2019.

Rajeev Agrawal. Sample mean based index policies by o(log n) regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27:1054 – 1078, 1995. URL https://api.semanticscholar.org/CorpusID:120313529.

Shipra Agrawal and Navin Goyal. Analysis of thompson sampling for the multi-armed bandit problem. In *Conference on learning theory*, pp. 39–1. JMLR Workshop and Conference Proceedings, 2012.

Shipra Agrawal and Navin Goyal. Thompson sampling for contextual bandits with linear payoffs. In *International conference on machine learning*, pp. 127–135. PMLR, 2013.

Jordan T. Ash, Cyril Zhang, Surbhi Goel, Akshay Krishnamurthy, and Sham M. Kakade. Anti-concentrated confidence bonuses for scalable exploration. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=RXQ-FPbQYVn.

Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.

Peter Auer and Ronald Ortner. Logarithmic online regret bounds for undiscounted reinforcement learning. *Advances in Neural Information Processing Systems*, 2006.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47:235–256, 2002.

Peter L Bartlett and Ambuj Tewari. Regal: A regularization based algorithm for reinforcement learning in weakly communicating mdps. *arXiv preprint arXiv:1205.2661*, 2012.

Marc G Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. *Advances in Neural Information Processing Systems*, 2016.

Ronen Brafman and Moshe Tennenholtz. R-max-a general polynomial time algorithm for near-optimal reinforcement learning. *The Journal of Machine Learning Research*, 2003.

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1lJJnR5Ym.

Qi Cai, Zhuoran Yang, Jason D Lee, and Zhaoran Wang. Neural Temporal-Difference Learning Converges to Global Optima. In *Advances in Neural Information Processing Systems*, 2019.

Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. *Advances in neural information processing systems*, 24, 2011.

Leshem Choshen, Lior Fox, and Yonatan Loewenstein. DORA the explorer: Directed outreaching reinforcement action-selection. *CoRR*, 2018.

Kamil Ciosek, Vincent Fortuin, Ryota Tomioka, Katja Hofmann, and Richard Turner. Conservative uncertainty estimation by fitting prior networks. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=BJlahxHYDS.

Bo Dai, Niao He, Yunpeng Pan, Byron Boots, and Le Song. Learning from conditional distributions via dual embeddings. In *Artificial Intelligence and Statistics*. PMLR, 2017.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Matteo Hessel, Ian Osband, Alex Graves, Volodymyr Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rywHCPkAW.

Aditya Gopalan, Shie Mannor, and Yishay Mansour. Thompson sampling for complex bandit problems. *arXiv preprint arXiv:1311.0466*, 2013.

R Grande, T Walsh, and J How. Sample efficient reinforcement learning with gaussian processes. In *International Conference on Machine Learning*, 2014.

Thomas Jaksch, Ronald Ortner, and Peter Auer. Near-optimal regret bounds for reinforcement learning. *The Journal of Machine Learning Research*, 2010.

David Janz, Jiri Hron, Przemysł aw Mazur, Katja Hofmann, José Miguel Hernández-Lobato, and Sebastian Tschiatschek. Successor uncertainties: Exploration and uncertainty in temporal difference learning. In *Advances in Neural Information Processing Systems*, 2019.

Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 1996.

Michael J Kearns and Satinder P Singh. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 2002.

Raksha Kumaraswamy, Matthew Schlegel, Adam White, and Martha White. Context-dependent upper-confidence bounds for directed exploration. *arXiv preprint arXiv:1811.06629*, 2018.

Tor Lattimore and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020.

Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *World Wide Web Conference*, 2010.

Cam Linke, Nadia M Ady, Martha White, Thomas Degris, and Adam White. Adapting behavior via intrinsic reward: A survey and empirical study. *Journal of artificial intelligence research*, 69: 1287–1332, 2020.

Marlos C Machado, Marc G Bellemare, and Michael Bowling. Count-based exploration with the successor representation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 5125–5133, 2020.

Jarryd Martin, Suraj Narayanan Sasikumar, Tom Everitt, and Marcus Hutter. Count-based exploration in feature space for reinforcement learning. In *International Joint Conference on Artificial IntelligenceI*, 2017.

Matthew K McLeod, Chunlok Lo, Matthew Kyle Schlegel, Andrew Jacobsen, Raksha Kumaraswamy, Martha White, and Adam M White. Continual auxiliary task learning. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=EpL9IFAMa3.

Nicolas Meuleau and Paul Bourgine. Exploration of multi-state environments - local measures and back-propagation of uncertainty. *Machine Learning*, 1999.

Abhishek Naik, Roshan Shariff, Niko Yasui, Hengshuai Yao, and Richard S. Sutton. Discounted reinforcement learning is not an optimization problem, 2019.

Gergely Neu and Ciara Pike-Burke. A Unifying View of Optimism in Episodic Reinforcement Learning. *arXiv.org*, 2020.

Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pp. 278–287. Citeseer, 1999.

Brendan O'Donoghue, Ian Osband, Remi Munos, and Vlad Mnih. The Uncertainty Bellman Equation and Exploration. In *International Conference on Machine Learning*, 2018.

I Osband, B Van Roy, and Z Wen. Generalization and exploration via randomized value functions. In *International Conference on Machine Learning*, 2016a.

Ian Osband and Benjamin Van Roy. Why is posterior sampling better than optimism for reinforcement learning? In *International Conference on Machine Learning*, 2017.

Ian Osband, Daniel Russo, and Benjamin Van Roy. (more) efficient reinforcement learning via posterior sampling. In *Advances in Neural Information Processing Systems*, 2013.

Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped dqn. In *Advances in Neural Information Processing Systems*, 2016b.

Ian Osband, John Aslanides, and Albin Cassirer. Randomized Prior Functions for Deep Reinforcement Learning. *NeurIPS*, 2018.

Ian Osband, Benjamin Van Roy, Daniel J Russo, Zheng Wen, et al. Deep exploration via randomized value functions. *J. Mach. Learn. Res.*, 20(124):1–62, 2019.

Ian Osband, Zheng Wen, Seyed Mohammad Asghari, Vikranth Dwaracherla, Morteza Ibrahimi, Xiuyuan Lu, and Benjamin Van Roy. Approximate Thompson Sampling via Epistemic Neural Networks. In *Uncertainty in Artificial Intelligence*, 2023.

Georg Ostrovski, Marc G Bellemare, Aäron van den Oord, and Rémi Munos. Count-based exploration with neural density models. In *International Conference on Machine Learning*, 2017.

Art B. Owen and Dean Eckles. Bootstrapping data arrays of arbitrary order. *The Annals of Applied Statistics*, 6(3), sep 2012. doi: 10.1214/12-aoas547. URL https://doi.org/10.1214%2F12-aoas547.

Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning*. PMLR, 2017.

Andrew Patterson, Adam White, and Martha White. A generalized projected bellman error for off-policy value estimation in reinforcement learning. *The Journal of Machine Learning Research*, 2022.

Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration. *arXiv.org*, 2017.

Daniel Russo and Benjamin Van Roy. Learning to optimize via posterior sampling. *Mathematics of Operations Research*, 39(4):1221–1243, 2014.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Ozan Sener and Silvio Savarese. Active learning for convolutional neural networks: A core-set approach. *arXiv preprint arXiv:1708.00489*, 2017.

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

Alexander L Strehl and Michael L Littman. An analysis of model-based Interval Estimation for Markov Decision Processes. *Journal of Computer and System Sciences*, December 2008.

Alexander L Strehl, Lihong Li, Eric Wiewiora, John Langford, and Michael L Littman. Pac model-free reinforcement learning. In *International Conference on Machine Learning*, 2006.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Csaba Szepesvári. *Algorithms for reinforcement learning*. Springer Nature, 2022.

Manel Tagorti and Bruno Scherrer. On the Rate of Convergence and Error Bounds for LSTD(λ). In *Proceedings of the 32nd International Conference on Machine Learning*. PMLR, 2015.

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

William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25, 1933.

Robert J Tibshirani and Bradley Efron. An introduction to the bootstrap. *Monographs on statistics and applied probability*, 57(1), 1993.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, 2016.

Yi Wan, Abhishek Naik, and Richard S Sutton. Learning and planning in average-reward markov decision processes. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 10653–10662. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/wan21a.html.

Yining Wang, Ruosong Wang, Simon S Du, and Akshay Krishnamurthy. Optimism in reinforcement learning with generalized linear function approximation. *arXiv preprint arXiv:1912.04136*, 2019.

Zhikang T Wang and Masahito Ueda. Convergent and efficient deep q network algorithm. In *International Conference on Learning Representations*, 2022.

Zheng Wen. *Efficient reinforcement learning with value function generalization*. Stanford University, 2014.

Martha White. Unifying task specification in reinforcement learning. In *International Conference on Machine Learning*, 2017.

Lin Zhao, Huaqing Xiong, and Yingbin Liang. Faster non-asymptotic convergence for double q-learning. In *Advances in Neural Information Processing Systems*, 2021.

Appendix A

A Discussion on Convergence Criteria for Value Bonuses

First let us discuss how the theory for LSTD applies to our setting. The result from (Tagorti & Scherrer, 2015, Corollary 1) bounds the error of the value function learned under LSTD to the true value function, assuming features are linearly independent (Assumption 1) and a mixing assumption for the environment and behavior policy (Assumption 2). This bound includes an error to the best linear solution, for infinite data, and the error between the best linear solution and the true value function. Because we are in the realizable case and the objective is convex for linear function approximation, the best linear solution is the true value function and in the limit of data the LSTD solution will reach this best linear solution. We can write this as a corollary of their result. Note their result is written by value functions, but automatically extends to action-value function by considering state-action features and stationary distribution $\mu_b(s,a) = \mu(s)\pi_b(a|s)$.

Corollary 1 (Corollary following from [Theorem 1)**.** *(Tagorti & Scherrer, 2015)] Assume we are given behavior policy π_b with stationary distribution μ and target policy π and the rewards are defined using a randomly sampled f_i from the set of linear functions on features $\phi(s,a)$ and the formula in Equation (3.2). Under Assumption 1 and 2 from (Tagorti & Scherrer, 2015), for a large enough number of samples T given by (Tagorti & Scherrer, 2015, Eq 6) (called n in their result), then f_{w_i} returned by LSTD satisfies*

$$\mathbb{E}_{s\sim\mu a\sim\pi_b(\cdot|s)}[(f_{w_i}(s,a) - f_i(s,a))^2] \leq O(1/\sqrt{T})$$

Now let us discuss how the work on neural TD applies to our setting (Cai et al., 2019). The result is proved for neural networks with a single hidden layer using a ReLU activation for the hidden layer, with the additional condition that the stationary distribution for the policy has a

bounded density over states and the stepsizes decrease at a rate of $1/\sqrt{t}$. This result immediately implies that our f_{w_i} should converge to f_i, because the global solution for this problem is f_i because it is in the value function class. We state this as a corollary of their result here, to be clear about how it applies.

Corollary 2 (Corollary following from [Theorem 4.6)**.** *(Cai et al., 2019)] Assume that 1) the policy π is fixed with stationary distribution μ, where $\mu(s)\pi(a|s)$ has bounded density across the space $x = (s, a)$ 2) the function class $\mathcal{F} = \{\frac{1}{\sqrt{m}}\sum_{j=1}^{m} b_j \max(x^\top w_j, 0)|W = (b_1, \ldots, b_m, w_1, \ldots, w_m), ||W - W(0)||_2 \leq B\}$ for $x = (s, a)$, $W(0)$ a point at which the weights are initialized in the algorithm and B some constant, 3) $\|x\|_2 = 1$ for all x and the rewards are defined using a randomly sampled f_i from $\mathcal{F}$ and the formula in Equation (3.2), and 4) the Neural TD algorithm (Algorithm 1 in (Cai et al., 2019)) is run for T steps with stepsize $\eta = \min((1-\gamma)/8, 1/\sqrt{T})$. Then the algorithm returns f_{w_i} that satisfies*

$$\mathbb{E}_{W\sim,\mu\pi}[(f_{w_i}(s, a) - f_i(s, a))^2] \leq \frac{O(B^2)}{(1-\gamma)^2\sqrt{T}} + O(B^2 m^{-1/2} + B^{5/2} m^{-1/4})$$

Proof. The result also requires that the reward magnitudes are all bounded, which they are by construction. Theorem 4.6 states that the outputted action-value function is bounded as above to the global optimum in the function class. Because $f_i(s, a)$ is in the function class, we know it is the global optimum. □

A.1 Experiment Details

A.1.1 Environment Details

Mountain Car is classic control problem of driving an underpowered car up a mountain. The original problem is set up as cost-to-goal, and here to frame it as a challenging exploration problem we offset the reward by 1, making it a sparse reward problem. The start state is sampled from the range $[-0.6, -0.4]$, which is the valley between two mountains, and the car starts with velocity zero.

Puddle World is a continuous state 2-dimensional world with $(x, y) \in [0, 1]^2$ with 2 intersecting puddles: (1) $[0.45, 0.4]$ to $[0.45, 0.8]$, and (2) $[0.1, 0.75]$ to $[0.45, 0.75]$. The puddles have a radius of 0.1 and the goal is the region $(x, y) \in [0.95, 1.0], [0.95, 1.0]$. The problem is cost-to-goal with additional penalty for when the agent is either puddle. The penalty for being in a puddle is proportional to the distance of the agent from the center of the puddle, i.e., negative reward for being close to the center. The agent chooses a direction of movement, resulting in displacement equal to $0.005 + \zeta, \zeta \sim N(\mu = 0, \sigma = 0.1)$ in the chosen direction. The starting positions for

episodes is uniformly sampled from $(x, y) \in [0.1, 0.3], [0.45, 0.65]$. High variance transitions coupled with high magnitude penalties make this a challenging exploration problem.

River Swim is a standard continuing exploration benchmark inspired by a fish trying to swim upriver, with high reward (+1) upstream which is difficult to reach and, a lower but still positive reward (+0.005), which is easily reachable downstream. The state space is continuous in $[0, 1]$, and the stochastic displacement is equal to $0.1 + \zeta, \zeta \sim N(\mu = 0, \sigma = 0.01)$ in the direction of the chosen action up or down. As swimming upstream is difficult, action up is stochastically switched to down. We also flip the observation such that the high reward is at observation 0 and the lower reward is at observation 1. We do this because we noticed that using random initialization with RelU activations would mostly result in a higher value for a higher input thus favouring the correct action in this case. The starting position is sampled uniformly in $[0.9, 1.0]$.

DeepSea is a hard exploration episodic grid world environment. In each state the agent can take two actions, left or right, which moves the agent down one row with column shifting based on left or right action. Collisions to the grid edges are handled by the agent staying in the same column but moving down one row. Since the agent can never access the states on the right side of the diagonal of the grid, the total number of states are thus $\frac{N \times (N+1)}{2}$. The most rewarding state is the state on the bottom right corner of the grid. To reach this the agent to take the action to go right every time. However, there is a penalty of $\frac{0.01}{N}$ for taking the action right, except for in this high rewarding state where the agent gets a reward of 1 for taking the right action. This makes it a very challenging environment. A policy that explores uniform randomly has the probability of 2^{-N} of reaching the goal state in each episode.

A.1.2 Algorithm Details

In the classic environments, every agent uses the same neural architecture, containing 2 non-linear layers with 50 nodes each and ReLU activation, followed by a linear output-layer. DQN-P, BDQN and all variants of VBE use target networks which are updated periodically after every τ steps. For DQN-P and BDQN we use $\tau = 4$ for all four classic environments. VBE and its variants use $\tau = 4$ for Mountain Car, Puddle World and River Swim, and $\tau = 64$ for DeepSea. We use a learning rate of $\alpha = 0.001$ and a discount factor of $\gamma = 0.99$. DQN-P, BDQN and VBE variants use an experience replay buffer that stores the most recent 50K transitions. The agent's parameters are updated after every step using a randomly sampled mini-batch of 128. We sweep the agents on bonus scales $c = [1.0, 3.0, 10.0]$, and ensemble sizes $k = [1, 2, 8, 20]$. The PPO version of ACB uses an ensemble size of $k = 128$, and RND uses a multi-layer neural network instead of an ensemble. Tables A.1, A.2, A.3 show the best performing sets of ensemble size k and bonus scale c for results in Sections 4.3, 4.4, 4.5.

	DeepSea
VBE	$k = 1, c = 1.0$
VBE-SL	$k = 20, c = 1.0$
DQN-P	$k = 1, c = 1.0$
BDQN	$k = 20, c = 1.0$
VB ACB	$k = 20, c = 1.0$
VB RND	$c = 1.0$

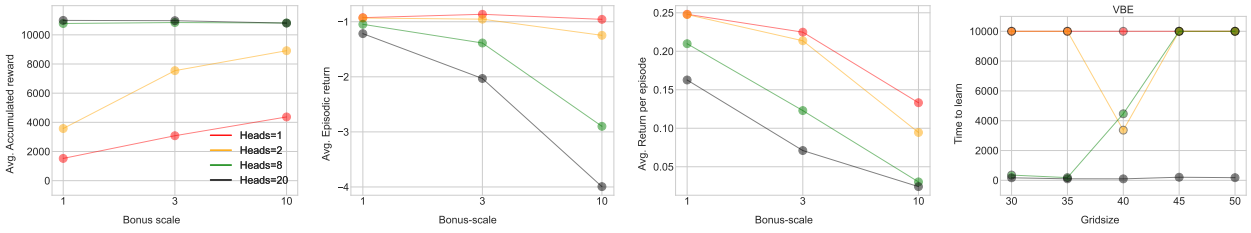
Table A.1: Ensemble size k and bonus scale c for agents in Figure 4.2

	River Swim	Puddle World	Mountain Car	DeepSea
VBE	$k = 20, c = 1.0$	$k = 1, c = 10.0$	$k = 2, c = 1.0$	$k = 20, c = 1.0$
DQN-P	$k = 1, c = 10.0$	$k = 1, c = 3.0$	$k = 1, c = 1.0$	$k = 1, c = 10.0$
BDQN	$k = 8, c = 10.0$	$k = 2, c = 1.0$	$k = 20, c = 1.0$	$k = 20, c = 10.0$
ACB ($k = 128$)	$c = 1.0$	$c = 3.0$	$c = 1.0$	$c = 10.0$
RND	$c = 1.0$	$c = 10.0$	$c = 10.0$	$c = 1.0$

Table A.2: Ensemble size k and bonus scale c for agents in Figure 4.3

A.2 Ensemble size × Bonus scale

In this section we show the effect that bonus scales and ensemble sizes have on the performance of the VBE in each of the four classic control environments. In Figure A.1 we show the average performance of VBE used in Section 4.4, for each environment. For Riverswim we see that the performance improves as the bonus scale and the ensemble size is increased. This makes sense as Riverswim is a hard exploration environment and requires more aggressive exploration. In Puddle world and Mountain car, we observe that increasing the bonus scale and the ensemble size harms the performance, since they do not rquire too much exploration. For Deepsea we only test a bonus scale of 1 with different ensemble sizes on different grid sizes. We can see that only an ensemble size of 20 works well on all grid sizes. For the single linear-layer agent we observe a similar patter in Figure A.2.

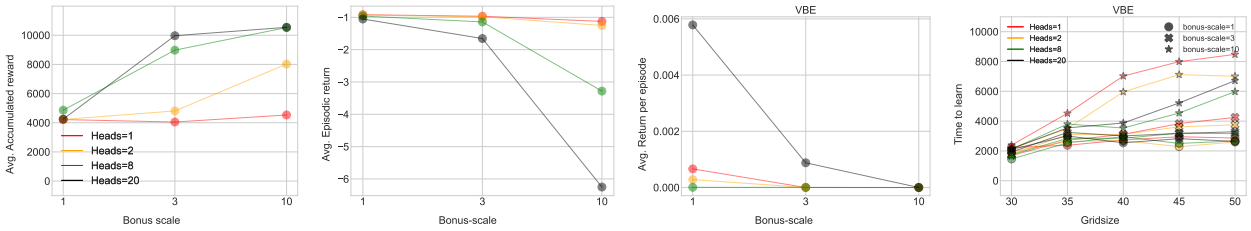


(a) River swim (b) Puddle world (c) Mountain car (d) Deepsea

Figure A.1: Shows the effect of different bonus scales and ensemble sizes across the classic control environments. For Deepsea, we only use a bonus scale of 1 and test different ensemble sizes.

	River Swim	Puddle World	Mountain Car	DeepSea
VBE	$k = 20, c = 1.0$	$k = 1, c = 10.0$	$k = 2, c = 1.0$	$k = 20, c = 1.0$
VB ACB	$k = 20, c = 10.0$	$k = 1, c = 1.0$	$k = 8, c = 1.0$	$k = 20, c = 1.0$
VB RND	$c = 10.0$	$c = 1.0$	$c = 1.0$	$c = 1.0$

Table A.3: Ensemble size k and bonus scale c for agents in Figure 4.5



(a) River swim (b) Puddle world (c) Mountain car (d) Deepsea

Figure A.2: Shows the effect of different bonus scales and ensemble sizes across the classic control environments. These results correspond to the single linear-layer agent.

A.3 Target policy experiments

In many cases it is straight forward to define the agent's behaviour policy depending on the problem at hand, for example, for exploration an agent can use an $\epsilon-$greedy policy, or use an upper confidence style bonus to select actions, lets call this the optimistic policy. However, in case of the general policy iteration setting it is not clear what the target policy should be to perform updates. Should the target policy also be optimistic/on-policy? Or should it be greedy only with respect to the value estimates/off-policy? The question becomes especially challenging to answer when there are multiple value functions or an ensemble of value functions, like in our case. What should be the target policy to update the RQFs? To investigate we performed a series of experiments using different types of target policies, i.e., optimistic, and greedy. The optimistic agent uses the optimistic policy to update the value head and the RQFs. The greedy agent uses the greedy target policy to update both the value head and the RQFs. The target policy for the value heads and the RQFs is consistent, this is because we want the RQFs to correspond to the value estimates. The intuition behind using an optimistic policy is that the target action-values it gives may have high prediction errors and updating the RQFs by bootstrapping off of values with high prediction error is likely to produce errors in current RQF predictions as well. This allows uncertainty to propagates allowing the agent to do directed exploration. In case of the greedy target policy, the uncertainty still propagates however, in this case the prediction errors don't play a role in the selection of action-values, for example, an action-value with a low value but a high prediction error may not be selected using a greedy target policy.

In Figure A.3 we show two different agents, optimistic and greedy, run on different grid sizes of

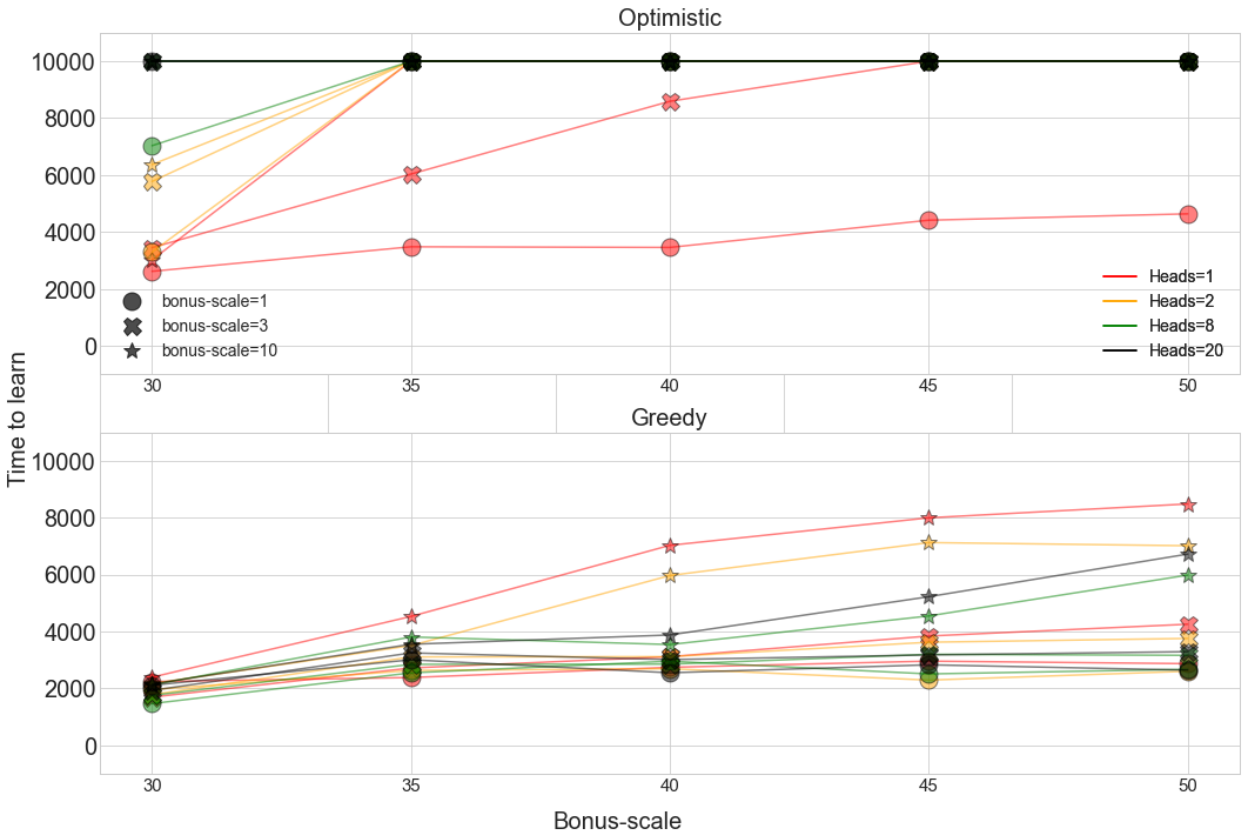
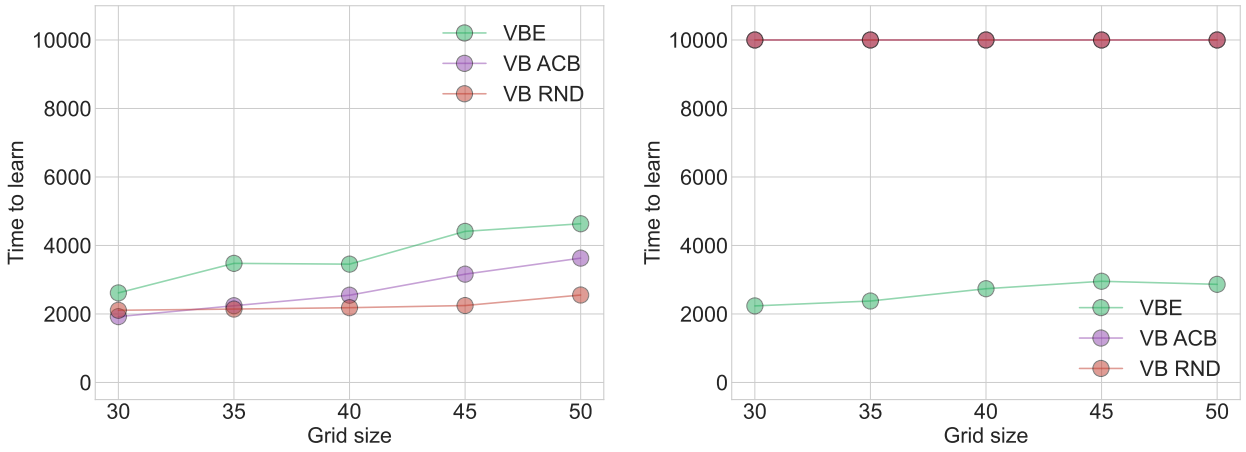


Figure A.3: Comparing Optimistic target policy (On-policy) with Greedy target policy (Off-policy) on different grid sizes of Deepsea.

DeepSea with different ensemble size, represented by color, and different bonus scales, represented by shapes. The agents use a single linear-layer for the value function and RQFs. We can see that the greedy agent generally performs better than the optimistic agent, which makes sense as the optimistic target policy can cause more exploration. The greedy agent performs well even with multiple RQFs, whereas optimistic agent fails to learn the optimal policy as the number of RQFs increase. We use the greedy target policy version of VBE in all the results of the main body.

In our experiments with VB with ACB and RND, we noticed a strange phenomenon, i.e., using an optimistic target policy allows VB ACB and VB RND to learn the optimal policy quickly on DeepsSa environments (Figure A.4), and using a greedy target policy for VB ACB/RND would cause the agents fail to learn the optimal policy (Figure 4.6d, 4.5d). This is interesting, as reward bonus methods do not provide optimism for unseen action-values, VB with ACB and RND should not be able to cover the entire state space based on random initialization. We found out that this happens because of the bias term in the linear layer, the momentum term in the optimizer and because the intrinsic value function is non-episodic. Using the optimistic target policy and with



(a) Optimistic target policy

(b) Greedy target policy

Figure A.4: In Figure A.4a the agents do on-policy (optimistic) updates. In Figure A.4b the agents do off-policy (greedy) updates. VB ACB/RND fail with the greedy policy, whereas with optimistic target policy they outperform VBE. VBE however, does well with a greedy policy compared to the optimistic one. These agents use a single linear-layer with bias term.

the help of momentum the bias term consistently increases, consequently the intrinsic action values start to increase. Since the bias-term is a shared parameter, the increase in its value provides the optimism for unseen action-values as well, this allows for the agent to cover the state space and thus learn the optimal policy. In case of the greedy target policy the intrinsic values do no increase, thus the agent fails to cover the state space. In Section 4.3, we show that if we use tabular features and a linear-layer without any bias term then VB ACB/RND fail to cover the state space. In this setting the agent's behaviour and target policy is governed by only the intrinsic value functions (on-policy), however it fails on account of not providing first-visit optimism.