Towards Informed Exploration for Deep Reinforcement Learning

by

Haoran Tang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Applied Mathematics

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor James Sethian, Co-chair
Professor Pieter Abbeel, Co-chair
Professor Trevor Darrell
Associate Professor Lin Lin

Fall 2019

Towards Informed Exploration for Deep Reinforcement Learning

Abstract

Towards Informed Exploration for Deep Reinforcement Learning

by

Haoran Tang

Doctor of Philosophy in Applied Mathematics

University of California, Berkeley

Professor James Sethian, Co-chair

Professor Pieter Abbeel, Co-chair

In this thesis, we discuss various techniques for improving exploration for deep reinforcement learning. We begin with a brief review of reinforcement learning (RL) and the fundamental exploration v.s. exploitation trade-off. Then we review how deep RL has improved upon classical methods and summarize six categories of the latest exploration methods for deep RL, in the order of increasing usage of prior information. We then explore representative works in three categories and discuss their strengths and weaknesses. The first category, represented by Soft Q-learning, uses entropy regularization to encourage exploration. The second category, represented by count-based exploration via hashing, maps states to hash codes for counting and assigns higher exploration bonuses to less-encountered states. The third category utilizes hierarchy and is represented by a modular architecture for RL agents to play StarCraft II. Finally, we conclude that exploration informed by prior knowledge is a promising research direction and suggest topics of potentially high impact.

Acknowledgments

I am extremely fortunate to study in Applied Math as a PhD student at UC Berkeley. I am very grateful to Prof. James Sethian, who introduced me to the exciting field of applied math, offered generous help in my PhD program application, and guided me through my earlier years of study. I would also like to thank Prof. Pieter Abbeel especially, who has offered constant support and advice to help me transition into the field of deep reinforcement learning and to make this thesis possible.

I am very thankful to Rocky Duan, Peter Chen, and Sergey Levine for their long-term guidance on my research. Thanks to my other collaborators for the work in this thesis, including Tuomas Haarnoja, Rein Houthooft, Davis Foote, Adam Stooke, Huazhe Xu, Dennis Lee, and Jeffrey O Zhang. Thanks to Lawrence Evans for serving on my quals committee, Trevor Darrell for serving on my dissertation committee, and Lin Lin for serving on both committees. Thanks to talented colleagues at UC Berkeley and DeepMind, including Carlos Florensa, Xinyang Geng, Tianhao Zhang, Yang Gao, Yi Wu, Keiran Paster, Yuhuai Wu, and Sasha Vezhnevets, for exchanging great research ideas. Thanks to my closest friends, Renyuan Xu, Difei Xiao, and Yingxin Chen for constant career and emotional guidance.

Finally, this thesis is dedicated to my parents Shuhuai Bi and Dengfeng Tang, for all the years of unconditional love and support.

Contents

Contents ii

1 Introduction **1**
1.1 Reinforcement Learning 1
1.2 The Exploration and Exploitation Trade-Off 1
1.3 Deep Reinforcement Learning 2
1.4 Techniques for Exploration in Deep Reinforcement Learning 3
1.5 Contributions of This Thesis 4

2 Reinforcement Learning with Deep Energy-Based Policies **6**
2.1 Introduction 6
2.2 Preliminaries 8
2.3 Training Expressive Energy-Based Models via Soft Q-Learning 10
2.4 Related Work 14
2.5 Experiments 15
2.6 Theoretical results 19
2.7 Implementation Details 23
2.8 Discussion 25

3 #Exploration: A Study of Count-Based Exploration for Deep Reinforcement Leaning **26**
3.1 Introduction 26
3.2 Methodology 27
3.3 Related Work 30
3.4 Experiments 31
3.5 A Case Study of Montezuma's Revenge 37
3.6 Discussion 37

4 Modular Architecture for StarCraft II with Deep Reinforcement Learning **40**
4.1 Introduction 40
4.2 Related Work 41
4.3 Modular Architecture 42
4.4 Training Procedure 47

4.5 Evaluation 48
4.6 Discussion 51

5 Conclusion **52**

Bibliography **54**

Chapter 1

Introduction

1.1 Reinforcement Learning

Reinforcement Learning (RL) is a trial-and-error framework for solving Markov Decision Processes (MDPs) [108]. An MDP consists of a state space $\mathcal{S}$, an action space $\mathcal{A}$, a transition function $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, \infty)$ s.t. $\int p(\mathbf{s}, \mathbf{a}, \mathbf{s}')\ d\mathbf{s}' = 1\ \forall \mathbf{s}, \mathbf{a}$ describing the probability of reaching a new state $\mathbf{s}'$ after taking action $\mathbf{a}$ at state $\mathbf{s}$, and a reward function $r(\mathbf{s}, \mathbf{a})$ received after the transition (some MDPs consider r stochastic). A (stochastic) policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, \infty)$ specifies a distribution over action $\mathbf{a}$ at state $\mathbf{s}$. A typical objective for an MDP is to maximize the discounted sum of rewards

$$\max_{\pi} \mathbb{E}_{\pi,p}\left[\sum_{t=0}^{\infty} \gamma^t r(\mathbf{s}_t, \mathbf{a}_t)\right] \tag{1.1}$$

RL usually refers to model-free RL, which assumes no or little prior knowledge of $\mathcal{S}$, $\mathcal{A}$, p, or r. In particular, the only available tool is an environment simulator that returns sampled trajectories (sequences of states and actions) for a given policy. Intuitively, a policy identifies patterns through trials, increases the probabilities of more rewarding actions, and repeats the process until it converges. Q-learning [108] is a classic example for this trial-and-error approach. As we shall see in Section 1.2, the generality of RL comes at the cost of sample inefficiency and usually requires careful algorithmic design.

Here we further define several useful terms. A trajectory is $\tau = \{(\mathbf{s}_t, \mathbf{a}_t)\}_{t=0}^{T}$. The Q-value for a policy π is $Q^\pi(\mathbf{s}, \mathbf{a}) = \mathbb{E}_{\pi,p}\left[\sum_{t=0}^{\infty} \gamma^t r(\mathbf{s}_t, \mathbf{a}_t) | \mathbf{s}_0 = \mathbf{s}, \mathbf{a}_0 = \mathbf{a}\right]$. The value for π is $V^\pi(\mathbf{s}) = \mathbb{E}_{\mathbf{a}\sim\pi(\mathbf{s},\cdot)}\left[Q^\pi(\mathbf{s}, \mathbf{a})\right]$. Q^* and V^* correspond to the optimal policy π^*.

1.2 The Exploration and Exploitation Trade-Off

During RL training, to guarantee convergence to the global optimum, each action at each state requires sufficient trials to accurately evaluate its outcome. But to accelerate convergence, the evaluation should be focused on (as currently perceived) more promising actions. The classic UCB1 algorithm summarizes such exploration v.s. exploitation trade-off in a multi-armed bandit (stateless MDP) setting.

$$\mathbf{a}_k = \arg\max_{\mathbf{a}} \left(\hat{r}_k(\mathbf{a}) + c\sqrt{\frac{2\log k}{N_{k-1}(\mathbf{a})}} \right) \tag{1.2}$$

where k is the number of iterations, $\hat{r}$ is the sample mean estimate of (stochastic) r, N is the number of times a has been tried, and c is a tunable positive constant. Here the reward estimate $\hat{r}_k$ emphasizes exploitation, while the remaining term favors under-explored actions, hence encourages exploration. A similar version is used in the Monte-Carlo Tree Search algorithm for the famous AlphaGo agent [102].

In MDPs with longer horizons, the trade-off becomes even more crucial. Since the total reward estimation has higher variance, exploitation is more likely to lead to local optimum. Moreover, exploration becomes harder due to the increasing number of branching factors. Numerous theoretical attempts have been made to address the trade-off in finite $\mathcal{S}$ and $\mathcal{A}$ settings, many resembling (1.2) in certain ways. They will be discussed in more detail in Section 3.3, but here let us preview a representative algorithm and understand the complexity.

MIBE-EB [106] is a count-based method for exploration. At each timestep k, it counts the occurrence N of each state-action pair, computes sample estimates $\hat{r}$, $\hat{p}$ of the reward and transition function, and solves an approximate Q-value that includes a bonus term for under-explored state-action pairs.

$$Q_k(\mathbf{s}, \mathbf{a}) = \hat{r}_k(\mathbf{s}, \mathbf{a}) + \gamma \sum_{\mathbf{s}'} \hat{p}_k(\mathbf{s}, \mathbf{a}, \mathbf{s}') \max_{\mathbf{a}'} Q_k(\mathbf{s}', \mathbf{a}') + \frac{\beta}{\sqrt{N_k(\mathbf{s}, \mathbf{a})}} \tag{1.3}$$

Then $\mathbf{a}_k = \arg\max_{\mathbf{a}} Q_k(\mathbf{s}_k, \mathbf{a})$ is executed at timestep k. This algorithm is "Provably Approximately Correct", in the sense that an MDP-dependent coefficient $\beta > 0$ exists such that with probability $(1-\delta)$ (across all runs of the algorithm), $V^{\pi_k}(\mathbf{s}_k) \geq V^*(\mathbf{s}_k) - \varepsilon$ for all but a polynomial in $(1/\varepsilon, 1/\delta, 1/(1-\gamma), |\mathcal{S}|, |\mathcal{A}|)$ timesteps. In particular, convergence is almost guaranteed in the limit, but near-optimal solution is unlikely guaranteed without massive sampling. This example shows why solving RL thoroughly is a fundamentally difficult.

Beware that most theoretical works focus on finite (and small) MDPs. Problems with continuous $\mathcal{S}$ and/or $\mathcal{A}$ present different opportunities and challenges. For example, imagery states are no longer treated distinct by minor differences in pixel values, but are compared by their visual semantics. On the other hand, the naive counting or sample mean estimates no longer work and different modeling techniques are required. For certain hard problems, one can even utilize prior knowledge to inform exploration. Section 1.4 will summarize such modern techniques.

1.3 Deep Reinforcement Learning

While traditional RL focuses on tabular or linear representations of $\mathcal{S}$ and $\mathcal{A}$, it is less effective against more complex problems, such as image observations. With the advance of deep learning [28], neural networks arise as powerful function approximators that can handle high-dimensional inputs and outputs. Soon they became popular candidates for value functions and policies, and have been successfully applied to playing Atari games [68], mastering the game of Go [102], and learning locomotion controllers [96, 60].

The most popular deep RL methods are policy gradient [96, 66, 94, 34] and Q-learning [69, 60, 33]. Unlike classical algorithms, they are distinctively model-free, scalable, and data-intensive. In fact, the ability to utilize massive data is the major reason for the success of deep RL.

1.4 Techniques for Exploration in Deep Reinforcement Learning

Techniques for exploration in deep RL are not fundamentally different from their classic counterparts. However, due to nonlinear function approximations, quantities such as $(\mathbf{s}, \mathbf{a})$ visitation frequency and variance of value estimates are not straightforward to obtain. Moreover, neural networks are able to generalize to different inputs, the analysis of which has been absent in traditional RL, and still remains unsolved even in deep learning.

Nevertheless, variants of classic methods still have empirical utility. There exists a spectrum of techniques, from the least to the most informed (i.e. armed with prior knowledge), which we shall summarize here.

1. **Entropy regularization**. Initiated in [66] and later formalized in [33], this method prevents early convergence to a deterministic policy by introducing policy entropy to the objective. Chapter 2 will discuss it in more detail.

2. **Novelty bonus**. These techniques characterize novel or under-explored events, such as new $\mathbf{s}$, $(\mathbf{s}, \mathbf{a})$, or $(\mathbf{s}, \mathbf{a}, \mathbf{s}')$, and assigns a bonus reward based on their novelty. This category of techniques has many names, such as count-based exploration [7], intrinsic motivation [41, 2], curiosity [84, 10], information gain [12], and others [26, 9]. Novelty bonus is by far the most popular research topic due to their effectiveness and generality.

3. **Hierarchy**. Hierarchical policies are thought to explore more efficiently, because the higher level can reason in longer horizons and in higher-level semantics. Successful examples include [55, 119, 22, 70].

4. **Reward shaping**. For certain challenging problems, such as MDPs with sparse rewards and long horizons, it can be easier to learn from a different but correlated reward. A famous example is presented in the OpenAI Dota project [78].

5. **Bootstrapping from demonstrations**. Sometimes the optimal solutions are so hard to search by RL that bootstrapping from demonstration is almost necessary. For example, self-play RL on StarCraft II was only able to learn naive rushing strategies. But by imitating expert actions and then learning from RL, the AlphaStar agent quickly surpassed human[120].

6. **Meta-exploration**. Meta-learning considers how to learn a new task quickly by leveraging knowledge from previously learned tasks. In particular, meta-exploration can acquire an exploration strategy from its learning experiences on other tasks. [17] and [32] have successful examples.

1.5 Contributions of This Thesis

In following chapters, we will see examples of detailed investigations into three types of exploration techniques: entropy regularization, novelty bonus, and hierarchy. The chosen order represents the author's growing interest towards more informed exploration (more prior knowledge). Chapter 5 will summarize lessons learned from the investigations and argue why informed exploration is important for solving challenging problems efficiently in deep RL.

The first contribution, titled "Reinforcement Learning with Deep Energy-Based Policies", studies how entropy regularization assists exploration. Drawing inspiration from stochastic optimal control, it proves that the optimal policy of the entropy-regularized objective has an energy form $\pi^*_{\text{soft}}(\mathbf{s}, \mathbf{a}) \propto_{\mathbf{a}} \exp(Q^*_{\text{soft}}(\mathbf{s}, \mathbf{a}))$. Moreover, Q^*_{soft} is the unique solution to the "soft Bellman equation", and therefore can be obtained from "soft Q-learning", analogous to the conventional Q-learning. In addition, π^*_{soft} uses a deep neural net to approximate the arbitrarily complex energy distribution, and is trained by Stein Variational Gradient Descent. Compared to the conventional Gaussian noise for greedy exploration, an energy-based policy assigns equal probabilities to equally rewarding actions, hence capturing various modes of good behavior. Experiments on continuous control tasks demonstrate its benefits on improved exploration and transfer learning. Videos of related experiments are available at `https://sites.google.com/view/softqlearning/home`. The source code can be accessed at `https://github.com/haarnoja/softqlearning`. A BAIR blog post summarizes key results along with new investigations: `https://bair.berkeley.edu/blog/2017/10/06/soft-q-learning`. This paper was published at 2017 International Conference on Machine Learning.

The second contribution, titled "#Exploration: A Study of Count-Based Exploration for Deep Reinforcement Leaning", studies how to implement count-based exploration via simple hash functions. The paper begins by reviewing classic count-based exploration methods with "provably approximately correct" guarantees. It then proposes bonus rewards of the form $\frac{\beta}{\sqrt{n(\phi(\mathbf{s}))}}$ to drive the agent towards under-explored states. The key is choosing a good hash function ϕ. It can be static and as simple as random projections. It can also be learned and represented by convolutional neural nets. Detailed experiments were performed on continuous control tasks from OpenAI Gym and challenging video games from Atari 2600. The simple hashing technique shows surprisingly competitive performance. A further investigation into Montezuma's Revenge shows how designing hash functions that only incorporate task-related information can boost scores dramatically. Videos of related experiments are available at `https://www.youtube.com/playlist?list=PLAd-UMX6FkBQdLNWtY8nH1-pzYJA_1T55`. This work was published at 2017 Conference on Neural Information Processing Systems.

The third contribution, titled “Modular Architecture for StarCraft II with Deep Reinforcement Learning”, studies how to simplify complex exploration problems by modularization, pretraining, and hierarchy. The game-play agent consists of five modules, each responsible for a relatively independent subtask. A module is first pretrained with other handcrafted modules, and then combined with other pretrained modules for fine-tuning. A hierarchical structure lets each module choose macro actions (predefined action sequences) instead of raw actions, which greatly reduces the search space. The agent is trained to play StarCraft II by competing against itself. Without ever seeing the test opponents, it is able to beat the “Harder” difficulty built-in bot with a 92% success rate. A video of the learned agent’s self-play can be viewed at `https://sites.google.com/view/modular-sc2-deeprl`. This work was published at The 14th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (2018).

Chapter 2

Reinforcement Learning with Deep Energy-Based Policies

2.1 Introduction

Deep reinforcement learning (deep RL) has emerged as a promising direction for autonomous acquisition of complex behaviors [68, 102], due to its ability to process complex sensory input [43] and to acquire elaborate behavior skills using general-purpose neural network representations [59]. Deep reinforcement learning methods can be used to optimize deterministic [60] and stochastic [96, 66] policies. However, most deep RL methods operate on the conventional deterministic notion of optimality, where the optimal solution, at least under full observability, is always a deterministic policy [108]. Although stochastic policies are desirable for exploration, this exploration is typically attained heuristically, for example by injecting noise [101, 60, 68] or initializing a stochastic policy with high entropy [47, 96, 66].

In some cases, we might actually prefer to learn stochastic behaviors. In this chapter, we explore two potential reasons for this: exploration in the presence of multimodal objectives, and compositionality attained via pretraining. Other benefits include robustness in the face of uncertain dynamics [127], imitation learning [128], and improved convergence and computational properties [30]. Multi-modality also has application in real robot tasks, as demonstrated in [16]. However, in order to learn such policies, we must define an objective that promotes stochasticity.

In which cases is a stochastic policy actually the optimal solution? As discussed in prior work, a stochastic policy emerges as the optimal answer when we consider the connection between optimal control and probabilistic inference [112]. While there are multiple instantiations of this framework, they typically include the cost or reward function as an additional factor in a factor graph, and infer the optimal conditional distribution over actions conditioned on states. The solution can be shown to optimize an entropy-augmented reinforcement learning objective or to correspond to the solution to a maximum entropy learning problem [115]. Intuitively, framing control as inference produces policies that aim to capture not only the single deterministic behavior that has the lowest cost, but the entire range of low-cost behaviors, explicitly maximizing the entropy of the corresponding policy. Instead of learning the best way to perform the task, the resulting policies try to learn *all* of the ways of performing the task. It should now be apparent why such policies might be preferred: if we can learn all of the ways that a given task might be performed, the resulting policy can serve as a good initialization for finetuning to a more specific behavior (e.g. first learning all the ways that a robot could move forward, and then using this as an initialization to learn separate running and bounding skills); a better exploration mechanism for seeking out the best mode in a multi-modal reward landscape; and a more robust behavior in the face of adversarial perturbations, where the ability to perform the same task in multiple different ways can provide the agent with more options to recover from perturbations.

Unfortunately, solving such maximum entropy stochastic policy learning problems in the general case is challenging. A number of methods have been proposed, including Z-learning [113], maximum entropy inverse RL [128], approximate inference using message passing [115], Ψ-learning [88], and G-learning [24], as well as more recent proposals in deep RL such as PGQ [74], but these generally operate either on simple tabular representations, which are difficult to apply to continuous or high-dimensional domains, or employ a simple parametric representation of the policy distribution, such as a conditional Gaussian. Therefore, although the policy is optimized to perform the desired skill in many different ways, the resulting distribution is typically very limited in terms of its representational power, even if the *parameters* of that distribution are represented by an expressive function approximator, such as a neural network.

How can we extend the framework of maximum entropy policy search to arbitrary policy distributions? In this chapter, we borrow an idea from energy-based models, which in turn reveals an intriguing connection between Q-learning, actor-critic algorithms, and probabilistic inference. In our method, we formulate a stochastic policy as a (conditional) energy-based model (EBM), with the energy function corresponding to the "soft" Q-function obtained when optimizing the maximum entropy objective. In high-dimensional continuous spaces, sampling from this policy, just as with any general EBM, becomes intractable. We borrow from the recent literature on EBMs to devise an approximate sampling procedure based on training a separate sampling network, which is optimized to produce unbiased samples from the policy EBM. This sampling network can then be used both for updating the EBM and for action selection. In the parlance of reinforcement learning, the sampling network is the actor in an actor-critic algorithm.

The principal contribution of this work is a tractable, efficient algorithm for optimizing arbitrary multimodal stochastic policies represented by energy-based models, as well as a discussion that relates this method to other recent algorithms in RL and probabilistic inference. In our experimental evaluation, we explore two potential applications of our approach. First, we demonstrate improved exploration performance in tasks with multi-modal reward landscapes, where conventional deterministic or unimodal methods are at high risk of falling into suboptimal local optima. Second, we explore how our method can be used to provide a degree of compositionality in reinforcement learning by showing that stochastic energy-based policies can serve as a much better initialization for learning new skills than either random policies or policies pretrained with conventional maximum reward objectives.

2.2 Preliminaries

In this section, we will define the reinforcement learning problem that we are addressing and briefly summarize the maximum entropy policy search objective. We will also present a few useful identities that we will build on in our algorithm, which will be presented in Section 2.3.

Maximum Entropy Reinforcement Learning

We will address learning of maximum entropy policies with approximate inference for reinforcement learning in continuous action spaces. Our reinforcement learning problem can be defined as policy search in an infinite-horizon Markov decision process (MDP), which consists of the tuple $(\mathcal{S}, \mathcal{A}, p_\mathbf{s}, r)$, The state space $\mathcal{S}$ and action space $\mathcal{A}$ are assumed to be continuous, and the state transition probability $p_\mathbf{s} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, \infty)$ represents the probability density of the next state $\mathbf{s}_{t+1} \in \mathcal{S}$ given the current state $\mathbf{s}_t \in \mathcal{S}$ and action $\mathbf{a}_t \in \mathcal{A}$. The environment emits a reward $r : \mathcal{S} \times \mathcal{A} \rightarrow [r_{\min}, r_{\max}]$ on each transition, which we will abbreviate as $r_t \triangleq r(\mathbf{s}_t, \mathbf{a}_t)$ to simplify notation. We will also use $\rho_\pi(\mathbf{s}_t)$ and $\rho_\pi(\mathbf{s}_t, \mathbf{a}_t)$ to denote the state and state-action marginals of the trajectory distribution induced by a policy $\pi(\mathbf{a}_t|\mathbf{s}_t)$.

Our goal is to learn a policy $\pi(\mathbf{a}_t|\mathbf{s}_t)$. We can define the standard reinforcement learning objective in terms of the above quantities as

$$\pi^*_{\text{std}} = \arg\max_\pi \sum_t \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\rho_\pi} \left[r(\mathbf{s}_t, \mathbf{a}_t) \right]. \tag{2.1}$$

Maximum entropy RL augments the reward with an entropy term, such that the optimal policy aims to maximize its entropy at each visited state:

$$\pi^*_{\text{MaxEnt}} = \arg\max_\pi \sum_t \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\rho_\pi} \left[r(\mathbf{s}_t, \mathbf{a}_t) + \alpha\mathcal{H}(\pi(\,\cdot\,|\mathbf{s}_t)) \right], \tag{2.2}$$

where α is an optional but convenient parameter that can be used to determine the relative importance of entropy and reward.[1] Optimization problems of this type have been explored in a number of prior works [48, 113, 128], which are covered in more detail in Section 2.4. Note that this objective differs qualitatively from the behavior of Boltzmann exploration [92] and PGQ [74], which greedily maximize entropy at the current time step, but do not explicitly optimize for policies that aim to reach *states* where they will have high entropy in the future. This distinction is crucial, since the maximum entropy objective can be shown to maximize the entropy of the entire trajectory distribution for the policy π, while the greedy Boltzmann exploration approach does not [128, 58]. As we will discuss in Section 3.4, this maximum entropy formulation has a number of benefits, such as improved exploration in multimodal problems and better pretraining for later adaptation.

If we wish to extend either the conventional or the maximum entropy RL objective to infinite horizon problems, it is convenient to also introduce a discount factor γ to ensure that the sum of expected rewards (and entropies) is finite. In the context of policy search algorithms, the use of a

[1]In principle, $1/\alpha$ can be folded into the reward function, eliminating the need for an explicit multiplier, but in practice, it is often convenient to keep α as a hyperparameter.

discount factor is actually a somewhat nuanced choice, and writing down the precise objective that is optimized when using the discount factor is non-trivial [111]. We defer the full derivation of the discounted objective to Section 2.6, since it is unwieldy to write out explicitly, but we will use the discount γ in the following derivations and in our final algorithm.

Soft Value Functions and Energy-Based Models

Optimizing the maximum entropy objective in (2.2) provides us with a framework for training stochastic policies, but we must still choose a representation for these policies. The choices in prior work include discrete multinomial distributions [74] and Gaussian distributions [88]. However, if we want to use a very general class of distributions that can represent complex, multimodal behaviors, we can instead opt for using general energy-based policies of the form

$$\pi(\mathbf{a}_t|\mathbf{s}_t) \propto \exp\left(-\mathcal{E}(\mathbf{s}_t, \mathbf{a}_t)\right), \tag{2.3}$$

where $\mathcal{E}$ is an energy function that could be represented, for example, by a deep neural network. If we use a universal function approximator for $\mathcal{E}$, we can represent any distribution $\pi(\mathbf{a}_t|\mathbf{s}_t)$. There is a close connection between such energy-based models and *soft* versions of value functions and Q-functions, where we set $\mathcal{E}(\mathbf{s}_t, \mathbf{a}_t) = -\frac{1}{\alpha}Q_{\text{soft}}(\mathbf{s}_t, \mathbf{a}_t)$ and use the following theorem:

Theorem 1. *Let the soft Q-function be defined by*

$$Q^*_{\text{soft}}(\mathbf{s}_t, \mathbf{a}_t) = r_t + \mathbb{E}_{(\mathbf{s}_{t+1},\dots)\sim\rho_\pi}\left[\sum_{l=1}^{\infty}\gamma^l\left(r_{t+l}+\alpha\mathcal{H}\left(\pi^*_{\text{MaxEnt}}(\,\cdot\,|\mathbf{s}_{t+l})\right)\right)\right],$$

and soft value function by

$$V^*_{\text{soft}}(\mathbf{s}_t) = \alpha\log\int_{\mathcal{A}}\exp\left(\frac{1}{\alpha}Q^*_{\text{soft}}(\mathbf{s}_t, \mathbf{a}')\right)d\mathbf{a}'. \tag{2.4}$$

Then the optimal policy for (2.2) is given by

$$\pi^*_{\text{MaxEnt}}(\mathbf{a}_t|\mathbf{s}_t)=\exp\left(\frac{1}{\alpha}(Q^*_{\text{soft}}(\mathbf{s}_t, \mathbf{a}_t)-V^*_{\text{soft}}(\mathbf{s}_t))\right). \tag{2.5}$$

Proof. See Section 2.6 as well as [127]. □

Theorem 1 connects the maximum entropy objective in (2.2) and energy-based models, where $\frac{1}{\alpha}Q_{\text{soft}}(\mathbf{s}_t, \mathbf{a}_t)$ acts as the negative energy, and $\frac{1}{\alpha}V_{\text{soft}}(\mathbf{s}_t)$ serves as the log-partition function. As with the standard Q-function and value function, we can relate the Q-function to the value function at a future state via a soft Bellman equation:

Theorem 2. *The soft Q-function in (2.4) satisfies the soft Bellman equation*

$$Q^*_{\text{soft}}(\mathbf{s}_t, \mathbf{a}_t) = r_t + \gamma\,\mathbb{E}_{\mathbf{s}_{t+1}\sim p_{\mathbf{s}}}\left[V^*_{\text{soft}}(\mathbf{s}_{t+1})\right], \tag{2.6}$$

where the soft value function V^*_{soft} is given by (2.4).

Proof. See Section 2.6, as well as [127]. □

The soft Bellman equation is a generalization of the conventional (hard) equation, where we can recover the more standard equation as $\alpha \to 0$, which causes (2.4) to approach a hard maximum over the actions. In the next section, we will discuss how we can use these identities to derive a Q-learning style algorithm for learning maximum entropy policies, and how we can make this practical for arbitrary Q-function representations via an approximate inference procedure.

2.3 Training Expressive Energy-Based Models via Soft Q-Learning

In this section, we will present our proposed reinforcement learning algorithm, which is based on the soft Q-function described in the previous section, but can be implemented via a tractable stochastic gradient descent procedure with approximate sampling. We will first describe the general case of soft Q-learning, and then present the inference procedure that makes it tractable to use with deep neural network representations in high-dimensional continuous state and action spaces. In the process, we will relate this Q-learning procedure to inference in energy-based models and actor-critic algorithms.

Soft Q-Iteration

We can obtain a solution to (2.6) by iteratively updating estimates of V^*_{soft} and Q^*_{soft}. This leads to a fixed-point iteration that resembles Q-iteration:

Theorem 3. *Soft Q-iteration. Let $Q_{\text{soft}}(\,\cdot\,,\,\cdot\,)$ and $V_{\text{soft}}(\,\cdot\,)$ be bounded and assume that $\int_{\mathcal{A}} \exp(\frac{1}{\alpha} Q_{\text{soft}}(\cdot, \mathbf{a}'))d\mathbf{a}' < \infty$ and that $Q^*_{\text{soft}} < \infty$ exists. Then the fixed-point iteration*

$$Q_{\text{soft}}(\mathbf{s}_t, \mathbf{a}_t) \leftarrow r_t + \gamma\, \mathbb{E}_{\mathbf{s}_{t+1} \sim p_{\mathbf{s}}}\left[V_{\text{soft}}(\mathbf{s}_{t+1})\right],\ \forall \mathbf{s}_t, \mathbf{a}_t \tag{2.7}$$

$$V_{\text{soft}}(\mathbf{s}_t) \leftarrow \alpha \log \int_{\mathcal{A}} \exp\left(\frac{1}{\alpha} Q_{\text{soft}}(\mathbf{s}_t, \mathbf{a}')\right) d\mathbf{a}',\ \forall \mathbf{s}_t \tag{2.8}$$

converges to Q^*_{soft} and V^*_{soft}, respectively.

Proof. See Section 2.6 as well as [24]. □

We refer to the updates in (2.7) and (2.8) as the soft Bellman backup operator that acts on the soft value function, and denote it by $\mathcal{T}$. The maximum entropy policy in (2.5) can then be recovered by iteratively applying this operator until convergence. However, there are several practicalities that need to be considered in order to make use of the algorithm. First, the soft Bellman backup cannot be performed exactly in continuous or large state and action spaces, and second, sampling from the energy-based model in (2.5) is intractable in general. We will address these challenges in the following sections.

Soft Q-Learning

This section discusses how the Bellman backup in Theorem 3 can be implemented in a practical algorithm that uses a finite set of samples from the environment, resulting in a method similar to Q-learning. Since the soft Bellman backup is a contraction (see Section 2.6), the optimal value function is the fixed point of the Bellman backup, and we can find it by optimizing for a Q-function for which the soft Bellman error $|\mathcal{T}Q - Q|$ is minimized at all states and actions. While this procedure is still intractable due to the integral in (2.8) and the infinite set of all states and actions, we can express it as a stochastic optimization, which leads to a stochastic gradient descent update procedure. We will model the soft Q-function with a function approximator with parameters θ and denote it as $Q^{\theta}_{\text{soft}}(\mathbf{s}_t, \mathbf{a}_t)$.

To convert Theorem 3 into a stochastic optimization problem, we first express the soft value function in terms of an expectation via importance sampling:

$$V^{\theta}_{\text{soft}}(\mathbf{s}_t) = \alpha \log \mathbb{E}_{q_{\mathbf{a}'}} \left[\frac{\exp\left(\frac{1}{\alpha} Q^{\theta}_{\text{soft}}(\mathbf{s}_t, \mathbf{a}')\right)}{q_{\mathbf{a}'}(\mathbf{a}')} \right], \tag{2.9}$$

where $q_{\mathbf{a}'}$ can be an arbitrary distribution over the action space. Second, by noting the identity $g_1(x) = g_2(x)\ \forall x \in \mathbb{X} \Leftrightarrow \mathbb{E}_{x\sim q}\left[(g_1(x) - g_2(x))^2\right] = 0$, where q can be any strictly positive density function on $\mathbb{X}$, we can express the soft Q-iteration in an equivalent form as minimizing

$$J_Q(\theta) = \mathbb{E}_{\mathbf{s}_t \sim q_{\mathbf{s}_t}, \mathbf{a}_t \sim q_{\mathbf{a}_t}} \left[\frac{1}{2} \left(\hat{Q}^{\bar{\theta}}_{\text{soft}}(\mathbf{s}_t, \mathbf{a}_t) - Q^{\theta}_{\text{soft}}(\mathbf{s}_t, \mathbf{a}_t) \right)^2 \right], \tag{2.10}$$

where $q_{\mathbf{s}_t}, q_{\mathbf{a}_t}$ are positive over $\mathcal{S}$ and $\mathcal{A}$ respectively, $\hat{Q}^{\bar{\theta}}_{\text{soft}}(\mathbf{s}_t, \mathbf{a}_t) = r_t + \gamma \mathbb{E}_{\mathbf{s}_{t+1}\sim p_{\mathbf{s}}}[V^{\bar{\theta}}_{\text{soft}}(\mathbf{s}_{t+1})]$ is a *target* Q-value, with $V^{\bar{\theta}}_{\text{soft}}(\mathbf{s}_{t+1})$ given by (2.9) and θ being replaced by the target parameters, $\bar{\theta}$.

This stochastic optimization problem can be solved approximately using stochastic gradient descent using sampled states and actions. While the sampling distributions $q_{\mathbf{s}_t}$ and $q_{\mathbf{a}_t}$ can be arbitrary, we typically use real samples from rollouts of the current policy $\pi(\mathbf{a}_t|\mathbf{s}_t) \propto \exp\left(\frac{1}{\alpha} Q^{\theta}_{\text{soft}}(\mathbf{s}_t, \mathbf{a}_t)\right)$. For $q_{\mathbf{a}'}$ we have more options. A convenient choice is a uniform distribution. However, this choice can scale poorly to high dimensions. A better choice is to use the current policy, which produces an unbiased estimate of the soft value as can be confirmed by substitution. This overall procedure yields an iterative approach that optimizes over the Q-values, which we summarize in Section 2.3.

However, in continuous spaces, we still need a tractable way to sample from the policy $\pi(\mathbf{a}_t|\mathbf{s}_t) \propto \exp\left(\frac{1}{\alpha} Q^{\theta}_{\text{soft}}(\mathbf{s}_t, \mathbf{a}_t)\right)$, both to take on-policy actions and, if so desired, to generate action samples for estimating the soft value function. Since the form of the policy is so general, sampling from it is intractable. We will therefore use an approximate sampling procedure, as discussed in the following section.

Approximate Sampling and Stein Variational Gradient Descent (SVGD)

In this section we describe how we can approximately sample from the soft Q-function. Existing approaches that sample from energy-based distributions generally fall into two categories: methods that use Markov chain Monte Carlo (MCMC) based sampling [92], and methods that learn a stochastic sampling network trained to output approximate samples from the target distribution

[126, 50]. Since sampling via MCMC is not tractable when the inference must be performed online (e.g. when executing a policy), we will use a sampling network based on Stein variational gradient descent (SVGD) [61] and amortized SVGD [122]. Amortized SVGD has several intriguing properties: First, it provides us with a stochastic sampling network that we can query for extremely fast sample generation. Second, it can be shown to converge to an accurate estimate of the posterior distribution of an EBM. Third, the resulting algorithm, as we will show later, strongly resembles actor-critic algorithm, which provides for a simple and computationally efficient implementation and sheds light on the connection between our algorithm and prior actor-critic methods.

Formally, we want to learn a state-conditioned stochastic neural network $\mathbf{a}_t = f^\phi(\xi; \mathbf{s}_t)$, parametrized by ϕ, that maps noise samples ξ drawn from a normal Gaussian, or other arbitrary distribution, into unbiased action samples from the target EBM corresponding to Q^θ_{soft}. We denote the induced distribution of the actions as $\pi^\phi(\mathbf{a}_t|\mathbf{s}_t)$, and we want to find parameters ϕ so that the induced distribution approximates the energy-based distribution in terms of the KL divergence

$$J_\pi(\phi; \mathbf{s}_t) = \mathrm{D}_{\mathrm{KL}}\left(\pi^\phi(\,\cdot\,|\mathbf{s}_t) \,\middle\|\, \exp\left(\frac{1}{\alpha}\left(Q^\theta_{\text{soft}}(\mathbf{s}_t, \,\cdot\,) - V^\theta_{\text{soft}}\right)\right)\right).$$

Suppose we "perturb" a set of independent samples $\mathbf{a}_t^{(i)} = f^\phi(\xi^{(i)}; \mathbf{s}_t)$ in appropriate directions $\Delta f^\phi(\xi^{(i)}; \mathbf{s}_t)$, the induced KL divergence can be reduced. Stein variational gradient descent [61] provides the most greedy directions as a functional

$$\Delta f^\phi(\,\cdot\,; \mathbf{s}_t) = \mathbb{E}_{\mathbf{a}_t \sim \pi^\phi}\left[\kappa(\mathbf{a}_t, f^\phi(\,\cdot\,; \mathbf{s}_t)) \nabla_{\mathbf{a}'} Q^\theta_{\text{soft}}(\mathbf{s}_t, \mathbf{a}')\big|_{\mathbf{a}'=\mathbf{a}_t} + \alpha \nabla_{\mathbf{a}'} \kappa(\mathbf{a}', f^\phi(\,\cdot\,; \mathbf{s}_t))\big|_{\mathbf{a}'=\mathbf{a}_t}\right],$$

where κ is a kernel function (typically Gaussian, see details in Section 2.7). To be precise, Δf^ϕ is the optimal direction in the reproducing kernel Hilbert space of κ, and is thus not strictly speaking the gradient of (2.11), but it turns out that we can set $\frac{\partial J_\pi}{\partial \mathbf{a}_t} \propto \Delta f^\phi$ as explained in [122]. With this assumption, we can use the chain rule and backpropagate the Stein variational gradient into the policy network according to

$$\frac{\partial J_\pi(\phi; \mathbf{s}_t)}{\partial \phi} \propto \mathbb{E}_\xi\left[\Delta f^\phi(\xi; \mathbf{s}_t) \frac{\partial f^\phi(\xi; \mathbf{s}_t)}{\partial \phi}\right], \tag{2.11}$$

and use any gradient-based optimization method to learn the optimal sampling network parameters. The sampling network f^ϕ can be viewed as an actor in an actor-critic algorithm. We will discuss this connection in Section 2.4, but first we will summarize our complete maximum entropy policy learning algorithm.

Algorithm Summary

To summarize, we propose the soft Q-learning algorithm for learning maximum entropy policies in continuous domains. The algorithm proceeds by alternating between collecting new experience from the environment, and updating the soft Q-function and sampling network parameters. The experience is stored in a replay memory buffer $\mathcal{D}$ as standard in deep Q-learning [69], and the parameters are updated using random minibatches from this memory. The soft Q-function updates use a delayed version of the target values [68]. For optimization, we use the ADAM [52] optimizer and empirical estimates of the gradients, which we denote by $\hat{\nabla}$. The exact formulae used to compute the gradient estimates is deferred to Section 2.7, which also discusses other implementation details, but we summarize an overview of soft Q-learning in algorithm 1.

Algorithm 1: Soft Q-learning

```
  $\theta, \phi \sim$ some initialization distributions.
  Assign target parameters: $\bar{\theta} \leftarrow \theta$, $\bar{\phi} \leftarrow \phi$.
  $\mathcal{D} \leftarrow$ empty replay memory.
for each epoch do
  for each $t$ do
    Collect experience
      Sample an action for $\mathbf{s}_t$ using $f^\phi$:
        $\mathbf{a}_t \leftarrow f^\phi(\xi; \mathbf{s}_t)$ where $\xi \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$.
      Sample next state from the environment:
        $\mathbf{s}_{t+1} \sim p_\mathbf{s}(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$.
      Save the new experience in the replay memory:
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$.
    Sample a minibatch from the replay memory
      $\{(\mathbf{s}_t^{(i)}, \mathbf{a}_t^{(i)}, r_t^{(i)}, \mathbf{s}_{t+1}^{(i)})\}_{i=0}^N \sim \mathcal{D}$.
    Update the soft Q-function parameters
      Sample $\{\mathbf{a}^{(i,j)}\}_{j=0}^M \sim q_{\mathbf{a}'}$ for each $\mathbf{s}_{t+1}^{(i)}$.
      Compute empirical soft values $\hat{V}_{\mathrm{soft}}^{\bar{\theta}}(\mathbf{s}_{t+1}^{(i)})$ in (2.9).
      Compute empirical gradient $\hat{\nabla}_\theta J_Q$ of (2.10).
      Update $\theta$ according to $\hat{\nabla}_\theta J_Q$ using ADAM.
    Update policy
      Sample $\{\xi^{(i,j)}\}_{j=0}^M \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ for each $\mathbf{s}_t^{(i)}$.
      Compute actions $\mathbf{a}_t^{(i,j)} = f^\phi(\xi^{(i,j)}, \mathbf{s}_t^{(i)})$.
      Compute $\Delta f^\phi$ using empirical estimate of (2.11).
      Compute empiricial estimate of (2.11): $\hat{\nabla}_\phi J_\pi$.
      Update $\phi$ according to $\hat{\nabla}_\phi J_\pi$ using ADAM.
  end for
  if epoch mod update_interval $= 0$ then
    Update target parameters: $\bar{\theta} \leftarrow \theta$, $\bar{\phi} \leftarrow \phi$.
  end if
end for
```

2.4 Related Work

Maximum entropy policies emerge as the solution when we cast optimal control as probabilistic inference. In the case of linear-quadratic systems, the mean of the maximum entropy policy is exactly the optimal deterministic policy [112], which has been exploited to construct practical path planning methods based on iterative linearization and probabilistic inference techniques [115]. In discrete state spaces, the maximum entropy policy can be obtained exactly. This has been explored in the context of linearly solvable MDPs [113] and, in the case of inverse reinforcement learning, MaxEnt IRL [128]. In continuous systems and continuous time, path integral control studies maximum entropy policies and maximum entropy planning [48]. In contrast to these prior methods, our work is focused on extending the maximum entropy policy search framework to high-dimensional continuous spaces and highly multimodal objectives, via expressive general-purpose energy functions represented by deep neural networks. A number of related methods have also used maximum entropy policy optimization as an intermediate step for optimizing policies under a standard expected reward objective [87, 73, 88, 24]. Among these, the work of [88] resembles ours in that it also makes use of a temporal difference style update to a soft Q-function. However, unlike this prior work, we focus on general-purpose energy functions with approximate sampling, rather than analytically normalizable distributions. A recent work [**liu2017stein**] also considers an entropy regularized objective, though the entropy is on policy parameters, not on sampled actions. Thus the resulting policy may not represent an arbitrarily complex multi-modal distribution with a single parameter. The form of our sampler resembles the stochastic networks proposed in recent work on hierarchical learning [22]. However this prior work uses a task-specific reward bonus system to encourage stochastic behavior, while our approach is derived from optimizing a general maximum entropy objective.

A closely related concept to maximum entropy policies is Boltzmann exploration, which uses the exponential of the standard Q-function as the probability of an action [46]. A number of prior works have also explored representing policies as energy-based models, with the Q-value obtained from an energy model such as a restricted Boltzmann machine (RBM) [92, 19, 82, 39]. Although these methods are closely related, they have not, to our knowledge, been extended to the case of deep network models, have not made extensive use of approximate inference techniques, and have not been demonstrated on the complex continuous tasks. More recently, [74] drew a connection between Boltzmann exploration and entropy-regularized policy gradient, though in a theoretical framework that differs from maximum entropy policy search: unlike the full maximum entropy framework, the approach of [74] only optimizes for maximizing entropy at the current time step, rather than planning for visiting future states where entropy will be further maximized. This prior method also does not demonstrate learning complex multi-modal policies in continuous action spaces.

Although we motivate our method as Q-learning, its structure resembles an actor-critic algorithm. It is particularly instructive to observe the connection between our approach and the deep deterministic policy gradient method (DDPG) [60], which updates a Q-function critic according to (hard) Bellman updates, and then backpropagates the Q-value gradient into the actor, similarly to NFQCA [35]. Our actor update differs only in the addition of the κ term. Indeed, without this term, our actor would estimate a maximum a posteriori (MAP) action, rather than capturing the entire EBM distribution. This suggests an intriguing connection between our method and DDPG: if we simply modify the DDPG critic updates to estimate soft Q-values, we recover the MAP variant of our method. Furthermore, this connection allows us to cast DDPG as simply an approximate Q-learning method, where the actor serves the role of an approximate maximizer. This helps explain the good performance of DDPG on off-policy data.

2.5 Experiments

Our experiments aim to answer the following questions: (1) Does our soft Q-learning method accurately capture a multi-modal policy distribution? (2) Can soft Q-learning with energy-based policies aid exploration for complex tasks that require tracking multiple modes? (3) Can a maximum entropy policy serve as a good initialization for finetuning on different tasks, when compared to pretraining with a standard deterministic objective? We compare our algorithm to DDPG [60], which has been shown to achieve better sample efficiency on the continuous control problems that we consider than other recent techniques such as REINFORCE [125], TRPO [96], and A3C [66]. This comparison is particularly interesting since, as discussed in Section 2.4, DDPG closely corresponds to a deterministic maximum a posteriori variant of our method. The detailed experimental setup can be found in Section 2.7. Videos of all experiments[2] and example source code[3] are available online.

Didactic Example: Multi-Goal Environment

In order to verify that amortized SVGD can correctly draw samples from energy-based policies of the form $\exp\left(Q^{\theta}_{\text{soft}}(s,a)\right)$, and that our complete algorithm can successful learn to represent multi-modal behavior, we designed a simple "multi-goal" environment, in which the agent is a 2D point mass trying to reach one of four symmetrically placed goals. The reward is defined as a mixture of Gaussians, with means placed at the goal positions. An optimal strategy is to go to an arbitrary goal, and the optimal maximum entropy policy should be able to choose each of the four goals at random. The final policy obtained with our method is illustrated in Figure 2.1. The Q-values indeed have complex shapes, being unimodal at $s = (-2, 0)$, convex at $s = (0, 0)$, and bimodal at $s = (2.5, 2.5)$. The stochastic policy samples actions closely following the energy landscape, hence learning diverse trajectories that lead to all four goals. In comparison, a policy trained with DDPG randomly commits to a single goal.

Learning Multi-Modal Policies for Exploration

Though not all environments have a clear multi-modal reward landscape as in the "multi-goal" example, multi-modality is prevalent in a variety of tasks. For example, a chess player might try various strategies before settling on one that seems most effective, and an agent navigating a maze may need to try various paths before finding the exit. During the learning process, it is often best to keep trying multiple available options until the agent is confident that one of them is the best

[2]https://sites.google.com/view/softqlearning/home
[3]https://github.com/haarnoja/softqlearning

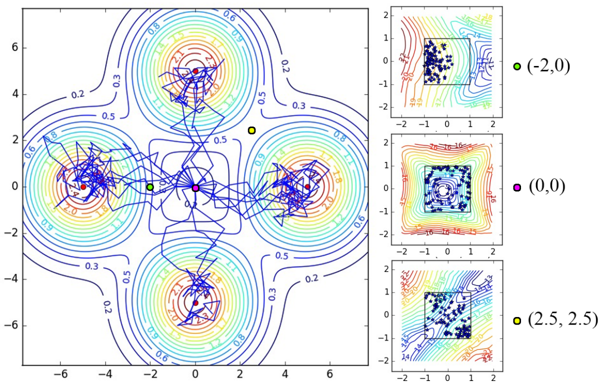
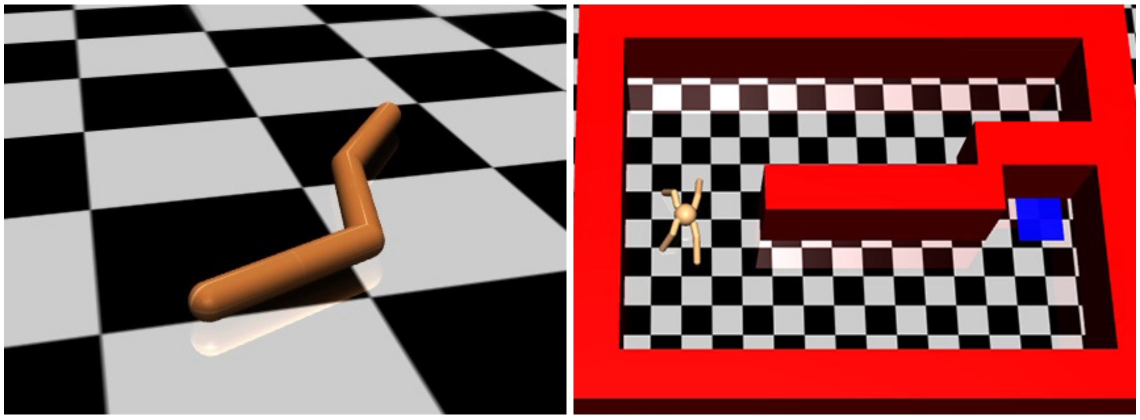


Figure 2.1: Illustration of the 2D multi-goal environment. Left: trajectories from a policy learned with our method (solid blue lines). The x and y axes correspond to 2D positions (states). The agent is initialized at the origin. The goals are depicted as red dots, and the level curves show the reward. Right: Q-values at three selected states, depicted by level curves (red: high values, blue: low values). The x and y axes correspond to 2D velocity (actions) bounded between -1 and 1. Actions sampled from the policy are shown as blue stars. Note that, in regions (e.g. $(2.5, 2.5)$) between the goals, the method chooses multimodal actions.



(a) Swimming snake (b) Quadrupedal robot

Figure 2.2: Simulated robots used in our experiments.

(similar to a bandit problem [56]). However, deep RL algorithms for continuous control typically use unimodal action distributions, which are not well suited to capture such multi-modality. As a consequence, such algorithms may prematurely commit to one mode and converge to suboptimal behavior.

To evaluate how maximum entropy policies might aid exploration, we constructed simulated continuous control environments where tracking multiple modes is important for success. The first experiment uses a simulated swimming snake (see Figure 2.2), which receives a reward equal to its speed along the x-axis, either forward or backward. However, once the swimmer swims far enough forward, it crosses a "finish line" and receives a larger reward. Therefore, the best learning strategy is to explore in both directions until the bonus reward is discovered, and then commit to swimming forward. As illustrated in Figure 2.3 in Figure 2.3, our method is able to recover this strategy, keeping track of both modes until the finish line is discovered. All stochastic policies eventually commit to swimming forward. The deterministic DDPG method shown in the comparison commits to a mode prematurely, with only 80% of the policies converging on a forward motion, and 20% choosing the suboptimal backward mode.

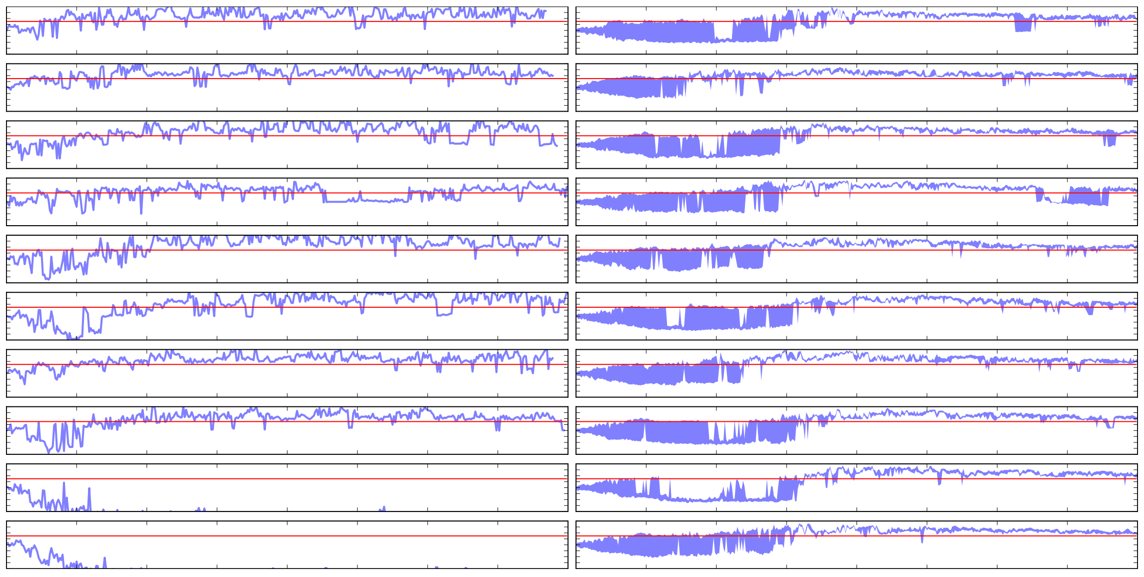
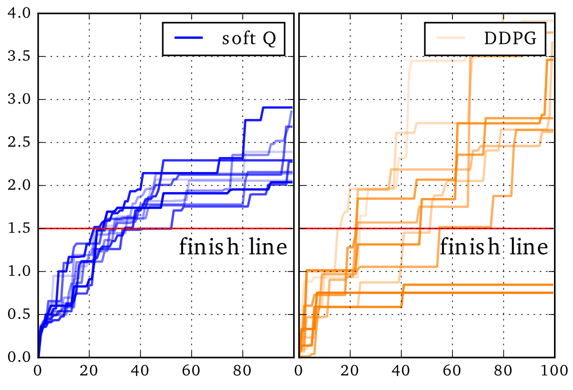
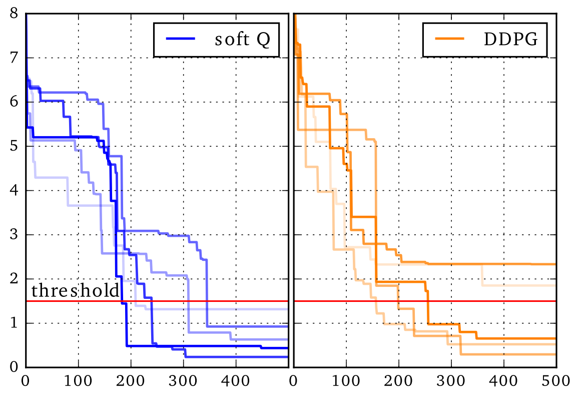


Figure 2.3: Forward swimming distance achieved by each policy. Each row is a policy with a unique random seed. x: training iteration, y: distance (positive: forward, negative: backward). Red line: the "finish line." The blue shaded region is bounded by the maximum and minimum distance (which are equal for DDPG). The plot shows that our method is able to explore equally well in both directions before it commits to the better one.

The second experiment studies a more complex task with a continuous range of equally good options prior to discovery of a sparse reward goal. In this task, a quadrupedal 3D robot (adapted from [95]) needs to find a path through a maze to a target position (see Figure 2.2). The reward function is a Gaussian centered at the target. The agent may choose either the upper or lower passage, which appear identical at first, but the upper passage is blocked by a barrier. Similar to the swimmer experiment, the optimal strategy requires exploring both directions and choosing the better one. Figure 2.4(b) compares the performance of DDPG and our method. The curves show the minimum distance to the target achieved so far and the threshold equals the minimum possible distance if the robot chooses the upper passage. Therefore, successful exploration means reaching below the threshold. All policies trained with our method manage to succeed, while only 60% policies trained with DDPG converge to choosing the lower passage.



(a) Swimmer (higher is better)



(b) Quadruped (lower is better)

Figure 2.4: Comparison of soft Q-learning and DDPG on the swimmer snake task and the quadrupedal robot maze task. (a) Shows the maximum traveled forward distance since the beginning of training for several runs of each algorithm; there is a large reward after crossing the finish line. (b) Shows our method was able to reach a low distance to the goal faster and more consistently. The different lines show the minimum distance to the goal since the beginning of training. For both domains, all runs of our method cross the threshold line, acquiring the more optimal strategy, while some runs of DDPG do not.

Accelerating Training on Complex Tasks with Pretrained Maximum Entropy Policies

A standard way to accelerate deep neural network training is task-specific initialization [28], where a network trained for one task is used as initialization for another task. The first task might be something highly general, such as classifying a large image dataset, while the second task might be more specific, such as fine-grained classification with a small dataset. Pretraining has also been explored in the context of RL [98]. However, in RL, near-optimal policies are often near-deterministic, which makes them poor initializers for new tasks. In this section, we explore how our energy-based policies can be trained with fairly broad objectives to produce an initializer for more quickly learning more specific tasks.

We demonstrate this on a variant of the quadrupedal robot task. The pretraining phase involves learning to locomote in an arbitrary direction, with a reward that simply equals the speed of the center of mass. The resulting policy moves the agent quickly to an randomly chosen direction. An overhead plot of the center of mass traces is shown above to illustrate this. This pretraining is similar in some ways to recent work on modulated controllers [40] and hierarchical models [22]. However, in contrast to these prior works, we do not require any task-specific high-level goal generator or reward.

Figure 2.6 also shows a variety of test environments that we used to finetune the running policy for a specific task. In the hallway environments, the agent receives the same reward, but the walls block sideways motion, so the optimal solution requires learning to run in a particular direction.

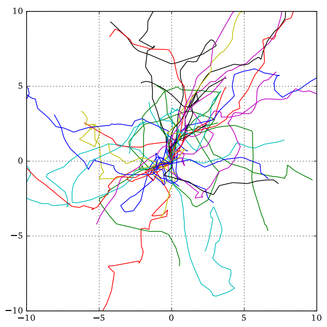


Figure 2.5: Center of mass plot after quadruped pretraining.

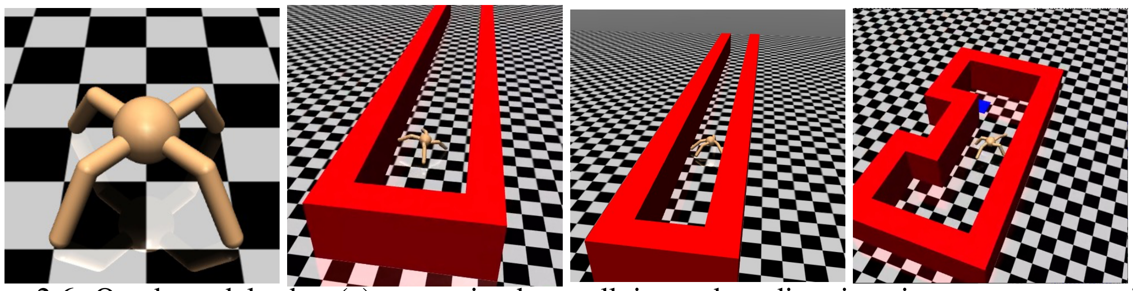


Figure 2.6: Quadrupedal robot (a) was trained to walk in random directions in an empty pretraining environment (details in Figure 2.7), and then finetuned on a variety of tasks, including a wide (b), narrow (c), and U-shaped hallway (d).

Narrow hallways require choosing a more specific direction, but also allow the agent to use the walls to funnel itself. The U-shaped maze requires the agent to learn a curved trajectory in order to arrive at the target, with the reward given by a Gaussian bump at the target location.

As illustrated in Figure 2.7, the pretrained policy explores the space extensively and in all directions. This gives a good initialization for the policy, allowing it to learn the behaviors in the test environments more quickly than training a policy with DDPG from a random initialization, as shown in Figure 2.8. We also evaluated an alternative pretraining method based on deterministic policies learned with DDPG. However, deterministic pretraining chooses an arbitrary but consistent direction in the training environment, providing a poor initialization for finetuning to a specific task, as shown in the results plots.

2.6 Theoretical results

Here we present proofs for the theorems that allow us to show that soft Q-learning leads to policy improvement with respect to the maximum entropy objective. First, we define a slightly more nuanced version of the maximum entropy objective that allows us to incorporate a discount factor. This definition is complicated by the fact that, when using a discount factor for policy gradient methods, we typically do not discount the state distribution, only the rewards. In that sense,

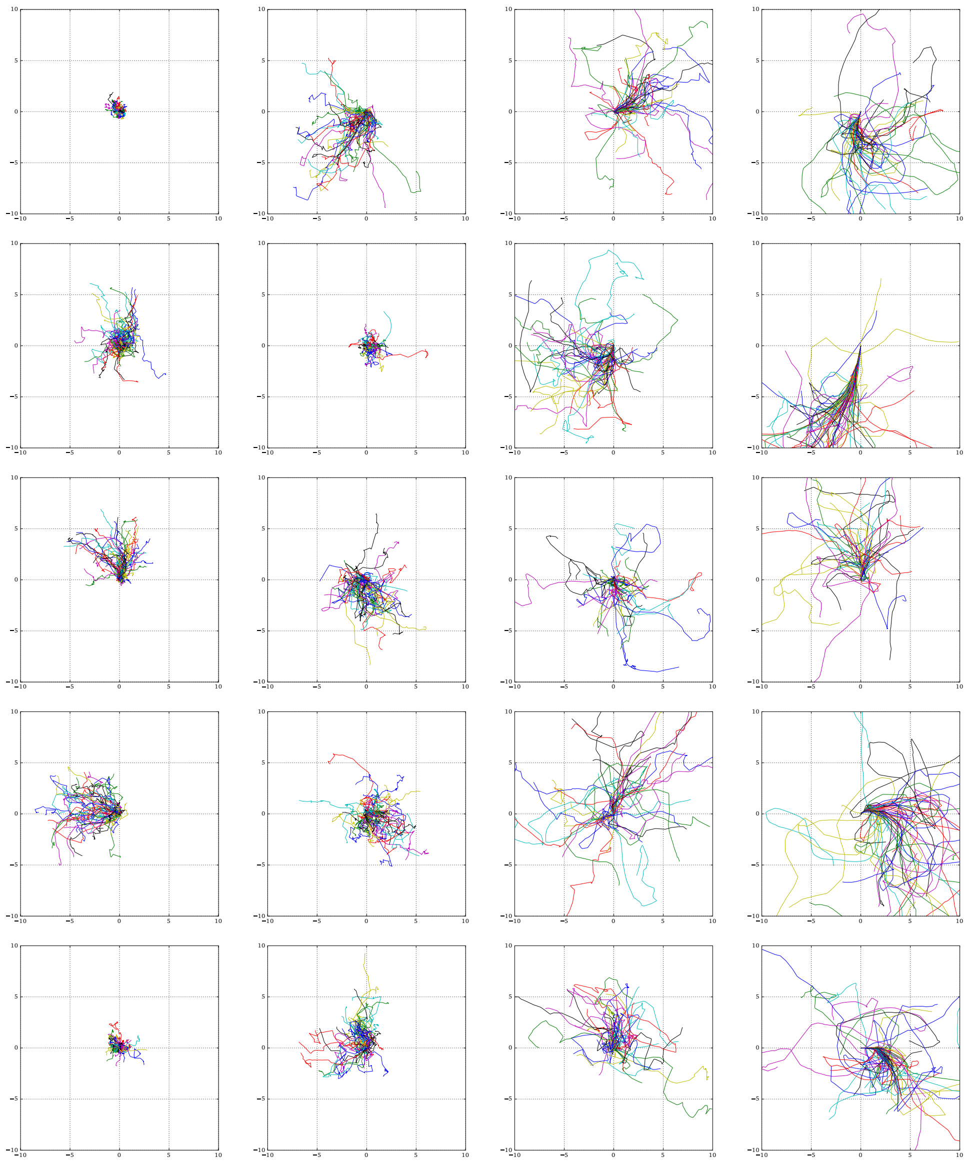


Figure 2.7: The plot shows trajectories of the quadrupedal robot during maximum entropy pre-training. The robot has diverse behavior and explores multiple directions. The four columns correspond to entropy coefficients $\alpha = 10, 1, 0.1, 0.01$ respectively. Different rows correspond to policies trained with different random seeds. The x and y axes show the x and y coordinates of the center-of-mass. As α decreases, the training process focuses more on high rewards, therefore exploring the training ground more extensively. However, low α also tends to produce less diverse behavior. Therefore the trajectories are more concentrated in the fourth column.

discounted policy gradients typically do not optimize the true discounted objective. Instead, they

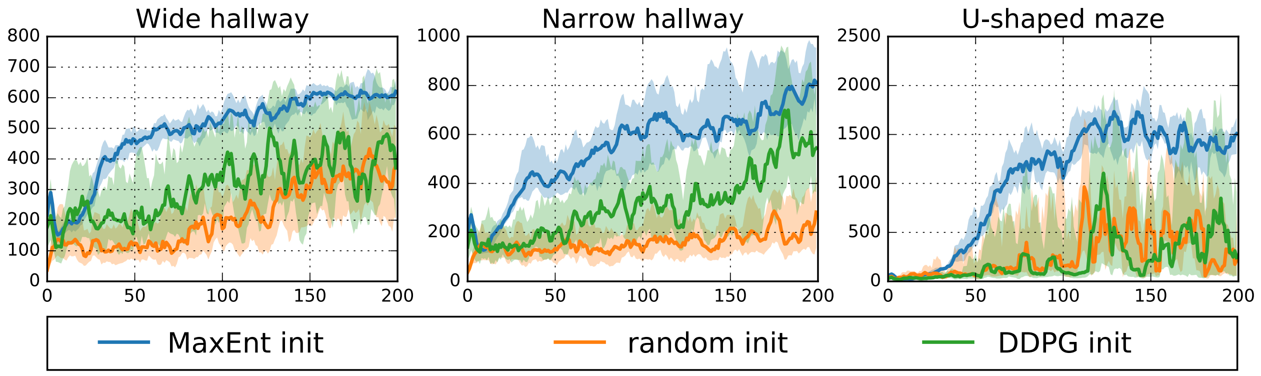


Figure 2.8: Performance in the downstream task with fine-tuning (MaxEnt) or training from scratch (DDPG). The x-axis shows the training iterations. The y-axis shows the average discounted return. Solid lines are average values over 10 random seeds. Shaded regions correspond to one standard deviation.

optimize average reward, with the discount serving to reduce variance, as discussed by [111]. However, for the purposes of the derivation, we can define the objective that *is* optimized under a discount factor as

$$\pi^*_{\text{MaxEnt}} = \arg\max_{\pi} \sum_t \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\rho_\pi} \left[\sum_{l=t}^{\infty} \gamma^{l-t} \, \mathbb{E}_{(\mathbf{s}_l,\mathbf{a}_l)} \left[r(\mathbf{s}_t, \mathbf{a}_t) + \alpha \mathcal{H}(\pi(\,\cdot\,|\mathbf{s}_t)) | \mathbf{s}_t, \mathbf{a}_t \right] \right].$$

This objective corresponds to maximizing the discounted expected reward and entropy for future states originating from every state-action tuple $(\mathbf{s}_t, \mathbf{a}_t)$ weighted by its probability ρ_π under the current policy. Note that this objective still takes into account the entropy of the policy at future states, in contrast to greedy objectives such as Boltzmann exploration or the approach proposed by [74].

We can now derive policy improvement results for soft Q-learning. We start with the definition of the soft Q-value Q^π_{soft} for any policy π as the expectation under π of the discounted sum of rewards and entropy :

$$Q^\pi_{\text{soft}}(\mathbf{s}, \mathbf{a}) \triangleq r_0 + \mathbb{E}_{\tau\sim\pi, \mathbf{s}_0=\mathbf{s}, \mathbf{a}_0=\mathbf{a}} \left[\sum_{t=1}^{\infty} \gamma^t (r_t + \mathcal{H}(\pi(\,\cdot\,|\mathbf{s}_t))) \right]. \tag{2.12}$$

Here, $\tau = (\mathbf{s}_0, \mathbf{a}_0, \mathbf{s}_1, \mathbf{a}_1, \ldots)$ denotes the trajectory originating at $(\mathbf{s}, \mathbf{a})$. Notice that for convenience, we set the entropy parameter α to 1. The theory can be easily adapted by dividing rewards by α.

The discounted maximum entropy policy objective can now be defined as

$$J(\pi) \triangleq \sum_t \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\rho_\pi} \left[Q^\pi_{\text{soft}}(\mathbf{s}_t, \mathbf{a}_t) + \alpha \mathcal{H}(\pi(\,\cdot\,|\mathbf{s}_t)) \right]. \tag{2.13}$$

The Maximum Entropy Policy

If the objective function is the expected discounted sum of rewards, the policy improvement theorem [108] describes how policies can be improved monotonically. There is a similar theorem we can derive for the maximum entropy objective:

Theorem 4. *(Policy improvement theorem) Given a policy π, define a new policy $\tilde{\pi}$ as*

$$\tilde{\pi}(\,\cdot\,|\mathbf{s}) \propto \exp\left(Q^{\pi}_{soft}(\mathbf{s},\,\cdot\,)\right), \quad \forall \mathbf{s}. \tag{2.14}$$

Assume that throughout our computation, Q is bounded and $\int \exp(Q(\mathbf{s},\mathbf{a}))\, d\mathbf{a}$ is bounded for any $\mathbf{s}$ (for both π and $\tilde{\pi}$). Then $Q^{\tilde{\pi}}_{soft}(\mathbf{s},\mathbf{a}) \geq Q^{\pi}_{soft}(\mathbf{s},\mathbf{a})\ \forall \mathbf{s},\mathbf{a}$.

The proof relies on the following observation: if one greedily maximize the sum of entropy and value with one-step look-ahead, then one obtains $\tilde{\pi}$ from π:

$$\mathcal{H}(\pi(\,\cdot\,|\mathbf{s})) + \mathbb{E}_{\mathbf{a}\sim\pi}\left[Q^{\pi}_{\text{soft}}(\mathbf{s},\mathbf{a})\right] \leq \mathcal{H}(\tilde{\pi}(\,\cdot\,|\mathbf{s})) + \mathbb{E}_{\mathbf{a}\sim\tilde{\pi}}\left[Q^{\pi}_{\text{soft}}(\mathbf{s},\mathbf{a})\right]. \tag{2.15}$$

The proof is straight-forward by noticing that

$$\mathcal{H}(\pi(\,\cdot\,|\mathbf{s})) + \mathbb{E}_{\mathbf{a}\sim\pi}\left[Q^{\pi}_{\text{soft}}(\mathbf{s},\mathbf{a})\right] = -\mathrm{D}_{\mathrm{KL}}\left(\pi(\,\cdot\,|\mathbf{s}) \parallel \tilde{\pi}(\,\cdot\,|\mathbf{s})\right) + \log \int \exp\left(Q^{\pi}_{\text{soft}}(\mathbf{s},\mathbf{a})\right)\, d\mathbf{a} \tag{2.16}$$

Then we can show that

$$\begin{aligned}
Q^{\pi}_{\text{soft}}(\mathbf{s},\mathbf{a}) &= \mathbb{E}_{\mathbf{s}_1}\left[r_0 + \gamma(\mathcal{H}(\pi(\,\cdot\,|\mathbf{s}_1)) + \mathbb{E}_{\mathbf{a}_1\sim\pi}\left[Q^{\pi}_{\text{soft}}(\mathbf{s}_1,\mathbf{a}_1)\right])\right] \\
&\leq \mathbb{E}_{\mathbf{s}_1}\left[r_0 + \gamma(\mathcal{H}(\tilde{\pi}(\,\cdot\,|\mathbf{s}_1)) + \mathbb{E}_{\mathbf{a}_1\sim\tilde{\pi}}\left[Q^{\pi}_{\text{soft}}(\mathbf{s}_1,\mathbf{a}_1)\right])\right] \\
&= \mathbb{E}_{\mathbf{s}_1}\left[r_0 + \gamma(\mathcal{H}(\tilde{\pi}(\,\cdot\,|\mathbf{s}_1)) + r_1)\right] + \gamma^2\,\mathbb{E}_{\mathbf{s}_2}\left[\mathcal{H}(\pi(\,\cdot\,|\mathbf{s}_2)) + \mathbb{E}_{\mathbf{a}_2\sim\pi}\left[Q^{\pi}_{\text{soft}}(\mathbf{s}_2,\mathbf{a}_2)\right]\right] \\
&\leq \mathbb{E}_{\mathbf{s}_1}\left[r_0 + \gamma(\mathcal{H}(\tilde{\pi}(\,\cdot\,|\mathbf{s}_1)) + r_1\right] + \gamma^2\,\mathbb{E}_{\mathbf{s}_2}\left[\mathcal{H}(\tilde{\pi}(\,\cdot\,|\mathbf{s}_2)) + \mathbb{E}_{\mathbf{a}_2\sim\tilde{\pi}}\left[Q^{\pi}_{\text{soft}}(\mathbf{s}_2,\mathbf{a}_2)\right]\right] \\
&= \mathbb{E}_{\mathbf{s}_1\,\mathbf{a}_2\sim\tilde{\pi},\mathbf{s}_2}\left[r_0 + \gamma(\mathcal{H}(\tilde{\pi}(\,\cdot\,|\mathbf{s}_1)) + r_1) + \gamma^2(\mathcal{H}(\tilde{\pi}(\,\cdot\,|\mathbf{s}_2)) + r_2)\right] \qquad (2.17) \\
&\quad + \gamma^3\,\mathbb{E}_{\mathbf{s}_3}\left[\mathcal{H}(\tilde{\pi}(\,\cdot\,|\mathbf{s}_3)) + \mathbb{E}_{\mathbf{a}_3\sim\tilde{\pi}}\left[Q^{\pi}_{\text{soft}}(\mathbf{s}_3,\mathbf{a}_3)\right]\right] \\
&\ \ \vdots \\
&\leq \mathbb{E}_{\tau\sim\tilde{\pi}}\left[r_0 + \sum_{t=1}^{\infty}\gamma^t(\mathcal{H}(\tilde{\pi}(\,\cdot\,|\mathbf{s}_t)) + r_t)\right] \\
&= Q^{\tilde{\pi}}_{\text{soft}}(\mathbf{s},\mathbf{a}). \qquad (2.18)
\end{aligned}$$

With Theorem 4, we start from an arbitrary policy π_0 and define the *policy iteration* as

$$\pi_{i+1}(\,\cdot\,|\mathbf{s}) \propto \exp\left(Q^{\pi_i}_{\text{soft}}(\mathbf{s},\,\cdot\,)\right). \tag{2.19}$$

Then $Q^{\pi_i}_{\text{soft}}(\mathbf{s},\mathbf{a})$ improves monotonically. Under certain regularity conditions, π_i converges to π_∞. Obviously, we have $\pi_\infty(\mathbf{a}|\mathbf{s}) \propto_{\mathbf{a}} \exp\left(Q^{\pi_\infty}(\mathbf{s},\mathbf{a})\right)$. Since any non-optimal policy can be improved this way, the optimal policy must satisfy this energy-based form. Therefore we have proven Theorem 1.

Soft Bellman Equation and Soft Value Iteration

Recall the definition of the soft value function:

$$V_{\text{soft}}^{\pi}(\mathbf{s}) \triangleq \log \int \exp\left(Q_{\text{soft}}^{\pi}(\mathbf{s}, \mathbf{a})\right)\, d\mathbf{a}. \tag{2.20}$$

Suppose $\pi(\mathbf{a}|\mathbf{s}) = \exp\left(Q_{\text{soft}}^{\pi}(\mathbf{s}, \mathbf{a}) - V_{\text{soft}}^{\pi}(\mathbf{s})\right)$. Then we can show that

$$\begin{aligned} Q_{\text{soft}}^{\pi}(\mathbf{s}, \mathbf{a}) &= r(\mathbf{s}, \mathbf{a}) + \gamma\, \mathbb{E}_{\mathbf{s}' \sim p_{\mathbf{s}}}\left[\mathcal{H}(\pi(\,\cdot\,|\mathbf{s}')) + \mathbb{E}_{\mathbf{a}' \sim \pi(\,\cdot\,|\mathbf{s}')}\left[Q_{\text{soft}}^{\pi}(\mathbf{s}', \mathbf{a}')\right]\right] \\ &= r(\mathbf{s}, \mathbf{a}) + \gamma\, \mathbb{E}_{\mathbf{s}' \sim p_{\mathbf{s}}}\left[V_{\text{soft}}^{\pi}(\mathbf{s}')\right]. \end{aligned} \tag{2.21}$$

This completes the proof of Theorem 2.

Finally, we show that the soft value iteration operator $\mathcal{T}$, defined as

$$\mathcal{T}Q(\mathbf{s}, \mathbf{a}) \triangleq r(\mathbf{s}, \mathbf{a}) + \gamma\, \mathbb{E}_{\mathbf{s}' \sim p_{\mathbf{s}}}\left[\log \int \exp Q(\mathbf{s}', \mathbf{a}')\, d\mathbf{a}'\right], \tag{2.22}$$

is a contraction. Then Theorem 3 follows immediately.

The following proof has also been presented by [24]. Define a norm on Q-values as $\|Q_1 - Q_2\| \triangleq \max_{\mathbf{s},\mathbf{a}} |Q_1(\mathbf{s}, \mathbf{a}) - Q_2(\mathbf{s}, \mathbf{a})|$. Suppose $\varepsilon = \|Q_1 - Q_2\|$. Then

$$\begin{aligned} \log \int \exp(Q_1(\mathbf{s}', \mathbf{a}'))\, d\mathbf{a}' &\le \log \int \exp(Q_2(\mathbf{s}', \mathbf{a}') + \varepsilon)\, d\mathbf{a}' \\ &= \log\left(\exp(\varepsilon) \int \exp Q_2(\mathbf{s}', \mathbf{a}')\, d\mathbf{a}'\right) \\ &= \varepsilon + \log \int \exp Q_2(\mathbf{a}', \mathbf{a}')\, d\mathbf{a}'. \end{aligned} \tag{2.23}$$

Similarly, $\log \int \exp Q_1(\mathbf{s}', \mathbf{a}')\, d\mathbf{a}' \ge -\varepsilon + \log \int \exp Q_2(\mathbf{s}', \mathbf{a}')\, d\mathbf{a}'$. Therefore $\|\mathcal{T}Q_1 - \mathcal{T}Q_2\| \le \gamma\varepsilon = \gamma\|Q_1 - Q_2\|$. So $\mathcal{T}$ is a contraction. As a consequence, only one Q-value satisfies the soft Bellman equation, and thus the optimal policy presented in Theorem 1 is unique.

2.7 Implementation Details

Hyperparameters

All tasks have a horizon of $T = 500$, except the multi-goal task, which uses $T = 20$. We add an additional termination condition to the quadrupedal 3D robot to discourage it from flipping over.

Throughout all experiments, we use the following parameters for both DDPG and soft Q-learning. The Q-values are updated using ADAM with learning rate 0.001. The DDPG policy and soft Q-learning sampling network use ADAM with a learning rate of 0.0001. The algorithm uses a replay pool of size one million. Training does not start until the replay pool has at least 10,000 samples. Every mini-batch has size 64. Each training iteration consists of 10000 time steps, and both the Q-values and policy / sampling network are trained at every time step. All experiments are run for 500 epochs, except that the multi-goal task uses 100 epochs and the fine-tuning tasks are trained for 200 epochs. Both the Q-value and policy / sampling network are neural networks comprised of two hidden layers, with 200 hidden units at each layer and ReLU nonlinearity. Both DDPG and soft Q-learning use additional OU Noise [116, 60] to improve exploration. The parameters are $\theta = 0.15$ and $\sigma = 0.3$. In addition, we found that updating the target parameters too frequently can destabilize training. Therefore we freeze target parameters for every 1000 time steps (except for the swimming snake experiment, which freezes for 5000 epochs), and then copy the current network parameters to the target networks directly ($\tau = 1$).

Soft Q-learning uses $K = M = 32$ action samples (see Section 2.7) to compute the policy update, except that the multi-goal experiment uses $K = M = 100$. The number of additional action samples to compute the soft value is $K_V = 50$. The kernel κ is a radial basis function, written as $\kappa(\mathbf{a}, \mathbf{a}') = \exp(-\frac{1}{h}\|\mathbf{a} - \mathbf{a}'\|_2^2)$, where $h = \frac{d}{2\log(M+1)}$, with d equal to the median of pairwise distance of sampled actions $\mathbf{a}_t^{(i)}$. Note that the step size h changes dynamically depending on the state $\mathbf{s}$, as suggested in [61].

The entropy coefficient α is 10 for multi-goal environment, and 0.1 for the swimming snake, maze, hallway (pretraining) and U-shaped maze (pretraining) experiments.

All fine-tuning tasks anneal the entropy coefficient α quickly in order to improve performance, since the goal during fine-tuning is to recover a near-deterministic policy on the fine-tuning task. In particular, α is annealed log-linearly to 0.001 within 20 epochs of fine-tuning. Moreover, the samples ξ are fixed to a set $\{\xi_i\}_{i=1}^{K_\xi}$ and K_ξ is reduced linearly to 1 within 20 epochs.

Computing the Policy Update

Here we explain in full detail how the policy update direction $\hat{\nabla}_\phi J_\pi$ in algorithm 1 is computed. We reuse the indices i, j in this section with a different meaning than in the body for the sake of providing a cleaner presentation.

Expectations appear in amortized SVGD in two places. First, SVGD approximates the optimal descent direction $\phi(\,\cdot\,)$ in Equation (2.11) with an empirical average over the samples $\mathbf{a}_t^{(i)} = f^\phi(\xi^{(i)})$. Similarly, SVGD approximates the expectation in Equation (2.11) with samples $\tilde{\mathbf{a}}_t^{(j)} = f^\phi(\tilde{\xi}^{(j)})$, which can be the same or different from $\mathbf{a}_t^{(i)}$. Substituting (2.11) into (2.11) and taking the gradient gives the empirical estimate

$$\hat{\nabla}_\phi J_\pi(\phi;\mathbf{s}_t)=\frac{1}{KM}\sum_{j=1}^{K}\sum_{i=1}^{M}\left(\kappa(\mathbf{a}_t^{(i)},\tilde{\mathbf{a}}_t^{(j)})\nabla_{\mathbf{a}'}Q_{\mathrm{soft}}(\mathbf{s}_t,\mathbf{a}')\Big|_{\mathbf{a}'=\mathbf{a}_t^{(i)}}+\nabla_{\mathbf{a}'}\kappa(\mathbf{a}',\tilde{\mathbf{a}}_t^{(j)})\Big|_{\mathbf{a}'=\mathbf{a}_t^{(i)}}\right)\nabla_\phi f^\phi(\tilde{\xi}^{(j)};\mathbf{s}_t).$$

Finally, the update direction $\hat{\nabla}_\phi J_\pi$ is the average of $\hat{\nabla}_\phi J_\pi(\phi; \mathbf{s}_t)$, where $\mathbf{s}_t$ is drawn from a mini-batch.

Computing the Density of Sampled Actions

Equation (2.9) states that the soft value can be computed by sampling from a distribution $q_{\mathbf{a}'}$ and that $q_{\mathbf{a}'}(\,\cdot\,) \propto \exp\left(\frac{1}{\alpha} Q_{\text{soft}}^{\phi}(\mathbf{s},\,\cdot\,)\right)$ is optimal. A direct solution is to obtain actions from the sampling network: $\mathbf{a}' = f^{\phi}(\xi'; \mathbf{s})$. If the samples ξ' and actions $\mathbf{a}'$ have the same dimension, and if the jacobian matrix $\frac{\partial \mathbf{a}'}{\partial \xi'}$ is non-singular, then the probability density is

$$q_{\mathbf{a}'}(\mathbf{a}') = p_{\xi}(\xi') \frac{1}{\left|\det\left(\frac{\partial \mathbf{a}'}{\partial \xi'}\right)\right|}. \tag{2.24}$$

In practice, the Jacobian is usually singular at the beginning of training, when the sampler f^{ϕ} is not fully trained. A simple solution is to begin with uniform action sampling and then switch to f^{ϕ} later, which is reasonable, since an untrained sampler is unlikely to produce better samples for estimating the partition function anyway.

2.8 Discussion

In summary, Soft Q-Learning improves exploration by explicitly sampling actions from $\exp(Q_{\text{soft}} - V_{\text{soft}})$, thus trying similarly rewarding actions with similar probabilities and eventually avoiding local minimums. However, in practice, sampling variation can still incorrectly favor less rewarding actions, which will cause early convergence through the snowball effect - more experiences are gathered using the worse actions, which in turn biases the Q-value estimate towards favoring them. Therefore the exploration vs exploitation conflict forces a trade-off between accurate value estimation and fast convergence.

To formalize the energy-based policy form, an entropy-regularized objective is proposed in place of total rewards. Despite its task-agnostic definition, it should be considered a modeling tool rather than an objective truth. Besides shaping the reward, one can easily change the optimal policy by remapping the action space or choosing a different temperature α. Therefore, it is up to the user to decide the form and degree of exploration.

Entropy regularization has limited power since it only involves properties of the reward function. In later chapters, we will study how to inform exploration by exploiting the state-space structure Chapter 3 and the decision hierarchy Chapter 4.

Chapter 3

#Exploration: A Study of Count-Based Exploration for Deep Reinforcement Leaning

3.1 Introduction

Reinforcement learning (RL) studies an agent acting in an initially unknown environment, learning through trial and error to maximize rewards. It is impossible for the agent to act near-optimally until it has sufficiently explored the environment and identified all of the opportunities for high reward, in all scenarios. A core challenge in RL is how to balance exploration—actively seeking out novel states and actions that might yield high rewards and lead to long-term gains; and exploitation—maximizing short-term rewards using the agent's current knowledge. While there are exploration techniques for finite MDPs that enjoy theoretical guarantees, there are no fully satisfying techniques for high-dimensional state spaces; therefore, developing more general and robust exploration techniques is an active area of research.

Most of the recent state-of-the-art RL results have been obtained using simple exploration strategies such as uniform sampling [68] and i.i.d./correlated Gaussian noise [96, 60]. Although these heuristics are sufficient in tasks with well-shaped rewards, the sample complexity can grow exponentially (with state space size) in tasks with sparse rewards [80]. Recently developed exploration strategies for deep RL have led to significantly improved performance on environments with sparse rewards. Bootstrapped DQN [81] led to faster learning in a range of Atari 2600 games by training an ensemble of Q-functions. Intrinsic motivation methods using pseudo-counts achieve state-of-the-art performance on Montezuma's Revenge, an extremely challenging Atari 2600 game [7]. Variational Information Maximizing Exploration (VIME, [41]) encourages the agent to explore by acquiring information about environment dynamics, and performs well on various robotic locomotion problems with sparse rewards. However, we have not seen a very simple and fast method that can work across different domains.

Some of the classic, theoretically-justified exploration methods are based on counting state-action visitations, and turning this count into a bonus reward. In the bandit setting, the well-known UCB algorithm of [56] chooses the action a_t at time t that maximizes $\hat{r}(a_t) + \sqrt{\frac{2\log t}{n(a_t)}}$ where $\hat{r}(a_t)$ is the estimated reward, and $n(a_t)$ is the number of times action a_t was previously chosen. In the MDP setting, some of the algorithms have similar structure, for example, Model Based Interval Estimation–Exploration Bonus (MBIE-EB) of [106] counts state-action pairs with a table $n(s,a)$ and adding a bonus reward of the form $\frac{\beta}{\sqrt{n(s,a)}}$ to encourage exploring less visited pairs. [53] show that the inverse-square-root dependence is optimal. MBIE and related algorithms assume that the augmented MDP is solved analytically at each timestep, which is only practical for small finite state spaces.

This chapter presents a simple approach for exploration, which extends classic counting-based methods to high-dimensional, continuous state spaces. We discretize the state space with a hash function and apply a bonus based on the state-visitation count. The hash function can be chosen to appropriately balance generalization across states, and distinguishing between states. We select problems from rllab [18] and Atari 2600 [6] featuring sparse rewards, and demonstrate near state-of-the-art performance on several games known to be hard for naïve exploration strategies. The main strength of the presented approach is that it is fast, flexible and complementary to most existing RL algorithms.

In summary, this chapter proposes a generalization of classic count-based exploration to high-dimensional spaces through hashing (Section 3.2); demonstrates its effectiveness on challenging deep RL benchmark problems and analyzes key components of well-designed hash functions (Section 3.4).

3.2 Methodology

Count-Based Exploration via Static Hashing

Our approach discretizes the state space with a hash function $\phi : \mathcal{S} \to \mathbb{Z}$. An exploration bonus $r^+ : \mathcal{S} \to \mathbb{R}$ is added to the reward function, defined as

$$r^+(s) = \frac{\beta}{\sqrt{n(\phi(s))}}, \tag{3.1}$$

where $\beta \in \mathbb{R}_{\geq 0}$ is the bonus coefficient. Initially the counts $n(\cdot)$ are set to zero for the whole range of ϕ. For every state $\mathbf{s}_t$ encountered at time step t, $n(\phi(\mathbf{s}_t))$ is increased by one. The agent is trained with rewards $(r + r^+)$, while performance is evaluated as the sum of rewards without bonuses.

Note that our approach is a departure from count-based exploration methods such as MBIE-EB since we use a state-space count $n(s)$ rather than a state-action count $n(s,a)$. State-action counts $n(s,a)$ are investigated in the Supplementary Material, but no significant performance gains over state counting could be witnessed. A possible reason is that the policy itself is sufficiently random to try most actions at a novel state.

Clearly the performance of this method will strongly depend on the choice of hash function ϕ. One important choice we can make regards the *granularity* of the discretization: we would like for

Algorithm 2: Count-based exploration through static hashing, using SimHash

```
1 Define state preprocessor $g : \mathcal{S} \to \mathbb{R}^D$
2 (In case of SimHash) Initialize $A \in \mathbb{R}^{k \times D}$ with entries drawn i.i.d. from the standard
    Gaussian distribution $\mathcal{N}(0, 1)$
3 Initialize a hash table with values $n(\cdot) \equiv 0$
4 for each iteration $j$ do
5     Collect a set of state-action samples $\{(s_m, a_m)\}_{m=0}^{M}$ with policy $\pi$
6     Compute hash codes through any LSH method, e.g., for SimHash,
        $\phi(s_m) = \operatorname{sgn}(Ag(s_m))$
7     Update the hash table counts $\forall m : 0 \leq m \leq M$ as $n(\phi(s_m)) \leftarrow n(\phi(s_m)) + 1$
8     Update the policy $\pi$ using rewards $\left\{ r(s_m, a_m) + \frac{\beta}{\sqrt{n(\phi(s_m))}} \right\}_{m=0}^{M}$ with any RL
        algorithm
```

"distant" states to be be counted separately while "similar" states are merged. If desired, we can incorporate prior knowledge into the choice of ϕ, if there would be a set of salient state features which are known to be relevant. A short discussion on this matter is given in the Supplementary Material.

Algorithm 2 summarizes our method. The main idea is to use locality-sensitive hashing (LSH) to convert continuous, high-dimensional data to discrete hash codes. LSH is a popular class of hash functions for querying nearest neighbors based on certain similarity metrics [4]. A computationally efficient type of LSH is SimHash [11], which measures similarity by angular distance. SimHash retrieves a binary code of state $s \in \mathcal{S}$ as

$$\phi(s) = \operatorname{sgn}(Ag(s)) \in \{-1, 1\}^k, \tag{3.2}$$

where $g : \mathcal{S} \to \mathbb{R}^D$ is an optional preprocessing function and A is a $k \times D$ matrix with i.i.d. entries drawn from a standard Gaussian distribution $\mathcal{N}(0, 1)$. The value for k controls the granularity: higher values lead to fewer collisions and are thus more likely to distinguish states.

Count-Based Exploration via Learned Hashing

When the MDP states have a complex structure, as is the case with image observations, measuring their similarity directly in pixel space fails to provide the semantic similarity measure one would desire. Previous work in computer vision [62, 15, 114] introduce manually designed feature representations of images that are suitable for semantic tasks including detection and classification. More recent methods learn complex features directly from data by training convolutional neural networks [54, 103, 38]. Considering these results, it may be difficult for a method such as SimHash to cluster states appropriately using only raw pixels.

Therefore, rather than using SimHash, we propose to use an autoencoder (AE) to learn meaningful hash codes in one of its hidden layers as a more advanced LSH method. This AE takes as input states s and contains one special dense layer comprised of D sigmoid functions. By rounding the sigmoid activations $b(s)$ of this layer to their closest binary number $\lfloor b(s) \rceil \in \{0,1\}^D$, any state s can be binarized. This is illustrated in Figure 3.1 for a convolutional AE.

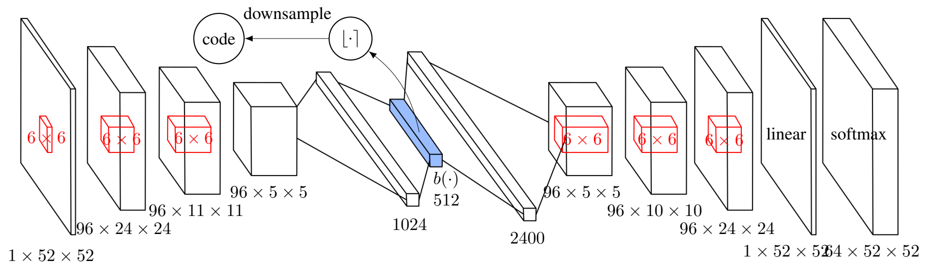


Figure 3.1: The autoencoder (AE) architecture for ALE; the solid block represents the dense sigmoidal binary code layer, after which noise $U(-a,a)$ is injected.

A problem with this architecture is that dissimilar inputs s_i, s_j can map to identical hash codes $\lfloor b(s_i) \rceil = \lfloor b(s_j) \rceil$, but the AE still reconstructs them perfectly. For example, if $b(s_i)$ and $b(s_j)$ have values 0.6 and 0.7 at a particular dimension, the difference can be exploited by deconvolutional layers in order to reconstruct s_i and s_j perfectly, although that dimension rounds to the same binary value. One can imagine replacing the bottleneck layer $b(s)$ with the hash codes $\lfloor b(s) \rceil$, but then gradients cannot be back-propagated through the rounding function. A solution is proposed by [29] and [90] is to inject uniform noise $U(-a,a)$ into the sigmoid activations. By choosing uniform noise with $a > \frac{1}{4}$, the AE is only capable of (always) reconstructing distinct state inputs $s_i \neq s_j$, if it has learned to spread the sigmoid outputs sufficiently far apart, $|b(s_i) - b(s_j)| > \epsilon$, in order to counteract the injected noise.

As such, the loss function over a set of collected states $\{s_i\}_{i=1}^N$ is defined as

$$L\left(\{s_n\}_{n=1}^N\right) = -\frac{1}{N}\sum_{n=1}^{N}\left[\log p(s_n) - \tfrac{\lambda}{K}\textstyle\sum_{i=1}^{D} \min\left\{\left(1-b_i(s_n)\right)^2, b_i(s_n)^2\right\}\right], \tag{3.3}$$

with $p(s_n)$ the AE output. This objective function consists of a negative log-likelihood term and a term that pressures the binary code layer to take on binary values, scaled by $\lambda \in \mathbb{R}_{\geq 0}$. The reasoning behind this latter term is that it might happen that for particular states, a certain sigmoid unit is never used. Therefore, its value might fluctuate around $\frac{1}{2}$, causing the corresponding bit in binary code $\lfloor b(s) \rceil$ to flip over the agent lifetime. Adding this second loss term ensures that an unused bit takes on an arbitrary binary value.

For Atari 2600 image inputs, since the pixel intensities are discrete values in the range $[0, 255]$, we make use of a pixel-wise softmax output layer [77] that shares weights between all pixels. The architectural details are described in the Supplementary Material and are depicted in Figure 3.1. Because the code dimension often needs to be large in order to correctly reconstruct the input, we apply a downsampling procedure to the resulting binary code $\lfloor b(s) \rceil$, which can be done through random projection to a lower-dimensional space via SimHash as in Eq. (3.2).

Algorithm 3: Count-based exploration using learned hash codes

```
1  Define state preprocessor $g : \mathcal{S} \rightarrow \{0,1\}^D$ as the binary code resulting from the
     autoencoder (AE)
2  Initialize $A \in \mathbb{R}^{k \times D}$ with entries drawn i.i.d. from the standard Gaussian distribution
     $\mathcal{N}(0,1)$
3  Initialize a hash table with values $n(\cdot) \equiv 0$
4  for each iteration $j$ do
5      Collect a set of state-action samples $\{(s_m, a_m)\}_{m=0}^{M}$ with policy $\pi$
6      Add the state samples $\{s_m\}_{m=0}^{M}$ to a FIFO replay pool $\mathcal{R}$
7      if $j$ mod $j_{\text{update}} = 0$ then
8          Update the AE loss function in Eq. (3.3) using samples drawn from the replay
             pool $\{s_n\}_{n=1}^{N} \sim \mathcal{R}$, for example using stochastic gradient descent
9      Compute $g(s_m) = \lfloor b(s_m) \rceil$, the $D$-dim rounded hash code for $s_m$ learned by the AE
10     Project $g(s_m)$ to a lower dimension $k$ via SimHash as $\phi(s_m) = \text{sgn}(Ag(s_m))$
11     Update the hash table counts $\forall m : 0 \leq m \leq M$ as $n(\phi(s_m)) \leftarrow n(\phi(s_m)) + 1$
12     Update the policy $\pi$ using rewards $\left\{ r(s_m, a_m) + \frac{\beta}{\sqrt{n(\phi(s_m))}} \right\}_{m=0}^{M}$ with any RL
         algorithm
```

On the one hand, it is important that the mapping from state to code needs to remain relatively consistent over time, which is nontrivial as the AE is constantly updated according to the latest data (Algorithm 3 line 8). A solution is to downsample the binary code to a very low dimension, or by slowing down the training process. On the other hand, the code has to remain relatively unique for states that are both distinct and close together on the image manifold. This is tackled both by the second term in Eq. (3.3) and by the saturating behavior of the sigmoid units. States already well represented by the AE tend to saturate the sigmoid activations, causing the resulting loss gradients to be close to zero, making the code less prone to change.

3.3 Related Work

Classic count-based methods such as MBIE [105], MBIE-EB and [53] solve an approximate Bellman equation as an inner loop before the agent takes an action [106]. As such, bonus rewards are propagated immediately throughout the state-action space. In contrast, contemporary deep RL algorithms propagate the bonus signal based on rollouts collected from interacting with environments, with value-based [68] or policy gradient-based [96, 66] methods, at limited speed. In addition, our proposed method is intended to work with contemporary deep RL algorithms, it differs from classical count-based method in that our method relies on visiting unseen states first, before the bonus reward can be assigned, making uninformed exploration strategies still a necessity at the beginning. Filling the gaps between our method and classic theories is an important direction of future research.

A related line of classical exploration methods is based on the idea of *optimism in the face of uncertainty* [8] but not restricted to using counting to implement "optimism", e.g., R-Max [8], UCRL [44], and E^3 [49]. These methods, similar to MBIE and MBIE-EB, have theoretical guarantees in tabular settings.

Bayesian RL methods [53, 31, 107, 27], which keep track of a distribution over MDPs, are an alternative to optimism-based methods. Extensions to continuous state space have been proposed by [85] and [80].

Another type of exploration is curiosity-based exploration. These methods try to capture the agent's surprise about transition dynamics. As the agent tries to optimize for surprise, it naturally discovers novel states. We refer the reader to [93] and [83] for an extensive review on curiosity and intrinsic rewards.

Several exploration strategies for deep RL have been proposed to handle high-dimensional state space recently. [41] propose VIME, in which information gain is measured in Bayesian neural networks modeling the MDP dynamics, which is used an exploration bonus. [104] propose to use the prediction error of a learned dynamics model as an exploration bonus. Thompson sampling through bootstrapping is proposed by [81], using bootstrapped Q-functions.

The most related exploration strategy is proposed by [7], in which an exploration bonus is added inversely proportional to the square root of a *pseudo-count* quantity. A state pseudo-count is derived from its log-probability improvement according to a density model over the state space, which in the limit converges to the empirical count. Our method is similar to pseudo-count approach in the sense that both methods are performing approximate counting to have the necessary generalization over unseen states. The difference is that a density model has to be designed and learned to achieve good generalization for pseudo-count whereas in our case generalization is obtained by a wide range of simple hash functions (not necessarily SimHash). Another interesting connection is that our method also implies a density model $\rho(s) = \frac{n(\phi(s))}{N}$ over all visited states, where N is the total number of states visited. Another method similar to hashing is proposed by [1], which clusters states and counts cluster centers instead of the true states, but this method has yet to be tested on standard exploration benchmark problems.

3.4 Experiments

Experiments were designed to investigate and answer the following research questions:

1. Can count-based exploration through hashing improve performance significantly across different domains? How does the proposed method compare to the current state of the art in exploration for deep RL?

2. What is the impact of learned or static state preprocessing on the overall performance when image observations are used?

To answer question 1, we run the proposed method on deep RL benchmarks (rllab and ALE) that feature sparse rewards, and compare it to other state-of-the-art algorithms. Question 2 is answered by trying out different image preprocessors on Atari 2600 games. Trust Region Policy Optimization (TRPO, [96]) is chosen as the RL algorithm for all experiments, because it can handle both discrete and continuous action spaces, can conveniently ensure stable improvement in the policy performance, and is relatively insensitive to hyperparameter changes. The hyperparameters settings are reported in the Supplementary Material.

Continuous Control

The rllab benchmark [18] consists of various control tasks to test deep RL algorithms. We selected several variants of the basic and locomotion tasks that use sparse rewards, as shown in Figure 3.2, and adopt the experimental setup as defined in [41]—a description can be found in the Supplementary Material. These tasks are all highly difficult to solve with naïve exploration strategies, such as adding Gaussian noise to the actions.

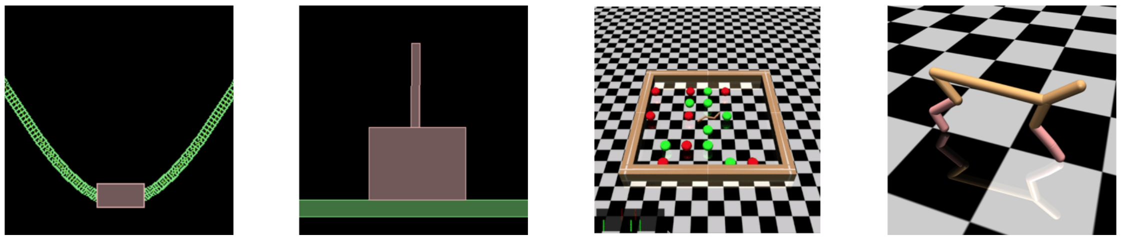
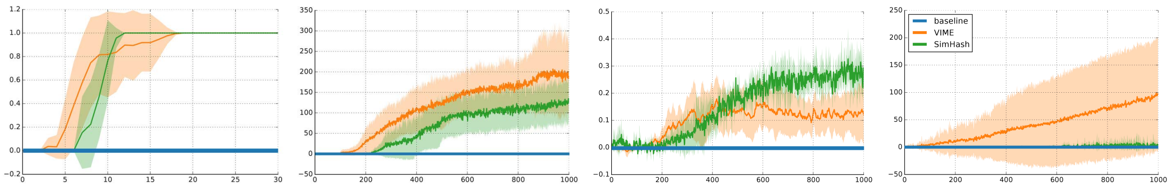


Figure 3.2: Illustrations of the rllab tasks used in the continuous control experiments, namely MountainCar, CartPoleSwingup, SimmerGather, and HalfCheetah; taken from [18].



(a) MountainCar (b) CartPoleSwingup (c) SwimmerGather (d) HalfCheetah

Figure 3.3: Mean average return of different algorithms on rllab tasks with sparse rewards. The solid line represents the mean average return, while the shaded area represents one standard deviation, over 5 seeds for the baseline and SimHash (the baseline curves happen to overlap with the axis).

Figure 3.3 shows the results of TRPO (baseline), TRPO-SimHash, and VIME [41] on the classic tasks MountainCar and CartPoleSwingup, the locomotion task HalfCheetah, and the hierarchical task SwimmerGather. Using count-based exploration with hashing is capable of reaching the goal in all environments (which corresponds to a nonzero return), while baseline TRPO with Gaussia n control noise fails completely. Although TRPO-SimHash picks up the sparse reward on

HalfCheetah, it does not perform as well as VIME. In contrast, the performance of SimHash is comparable with VIME on MountainCar, while it outperforms VIME on SwimmerGather.

Arcade Learning Environment

The Arcade Learning Environment (ALE, [6]), which consists of Atari 2600 video games, is an important benchmark for deep RL due to its high-dimensional state space and wide variety of games. In order to demonstrate the effectiveness of the proposed exploration strategy, six games are selected featuring long horizons while requiring significant exploration: Freeway, Frostbite, Gravitar, Montezuma's Revenge, Solaris, and Venture. The agent is trained for 500 iterations in all experiments, with each iteration consisting of $0.1\,\mathrm{M}$ steps (the TRPO batch size, corresponds to $0.4\,\mathrm{M}$ frames). Policies and value functions are neural networks with identical architectures to [66]. Although the policy and baseline take into account the previous four frames, the counting algorithm only looks at the latest frame.

BASS To compare with the autoencoder-based learned hash code, we propose using Basic Abstraction of the ScreenShots (BASS, also called Basic; see [6]) as a static preprocessing function g. BASS is a hand-designed feature transformation for images in Atari 2600 games. BASS builds on the following observations specific to Atari: 1) the game screen has a low resolution, 2) most objects are large and monochrome, and 3) winning depends mostly on knowing object locations and motions. We designed an adapted version of BASS[1], that divides the RGB screen into square cells, computes the average intensity of each color channel inside a cell, and assigns the resulting values to bins that uniformly partition the intensity range $[0, 255]$. Mathematically, let C be the cell size (width and height), B the number of bins, (i, j) cell location, (x, y) pixel location, and z the channel, then

$$\mathrm{feature}(i, j, z) = \left\lfloor \tfrac{B}{255C^2} \textstyle\sum_{(x,y)\in\ \mathrm{cell}(i,j)} I(x, y, z) \right\rfloor . \tag{3.4}$$

Afterwards, the resulting integer-valued feature tensor is converted to an integer hash code ($\phi(s_t)$ in Line 6 of Algorithm 2). A BASS feature can be regarded as a miniature that efficiently encodes object locations, but remains invariant to negligible object motions. It is easy to implement and introduces little computation overhead. However, it is designed for generic Atari game images and may not capture the structure of each specific game very well.

We compare our results to double DQN [36], dueling network [123], A3C+ [7], double DQN with pseudo-counts [7], Gorila [72], and DQN Pop-Art [37] on the "null op" metric[2]. We show training curves in Figure 3.4 and summarize all results in Table 1. Surprisingly, TRPO-pixel-SimHash already outperforms the baseline by a large margin and beats the previous best result on Frostbite. TRPO-BASS-SimHash achieves significant improvement over TRPO-pixel-SimHash on

[1]The original BASS exploits the fact that at most 128 colors can appear on the screen. Our adapted version does not make this assumption.

[2]The agent takes no action for a random number (within 30) of frames at the beginning of each episode.

Table 3.1: Atari 2600: average total reward after training for $50\,\mathrm{M}$ time steps. Boldface numbers indicate best results. Italic numbers are the best among our methods.

	Freeway	Frostbite	Gravitar	Montezuma	Solaris	Venture
TRPO (baseline)	16.5	2869	486	0	2758	121
TRPO-pixel-SimHash	31.6	4683	468	0	2897	263
TRPO-BASS-SimHash	28.4	3150	*604*	*238*	1201	*616*
TRPO-AE-SimHash	***33.5***	***5214***	482	75	*4467*	445
Double-DQN	33.3	1683	412	0	3068	98.0
Dueling network	0.0	4672	588	0	2251	497
Gorila	11.7	605	**1054**	4	N/A	**1245**
DQN Pop-Art	33.4	3469	483	0	**4544**	1172
A3C+	27.3	507	246	142	2175	0
pseudo-count	29.2	1450	–	**3439**	–	369

Montezuma's Revenge and Venture, where it captures object locations better than other methods.[3] TRPO-AE-SimHash achieves near state-of-the-art performance on Freeway, Frostbite and Solaris.

As observed in Table 1, preprocessing images with BASS or using a learned hash code through the AE leads to much better performance on Gravitar, Montezuma's Revenge and Venture. Therefore, a static or adaptive preprocessing step can be important for a good hash function.

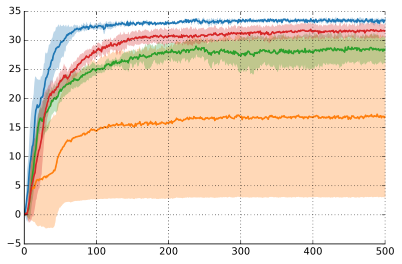
In conclusion, our count-based exploration method is able to achieve remarkable performance gains even with simple hash functions like SimHash on the raw pixel space. If coupled with domain-dependent state preprocessing techniques, it can sometimes achieve far better results.

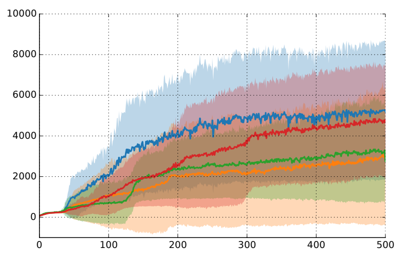
A reason why our proposed method does not achieve state-of-the-art performance on all games is that TRPO does not reuse off-policy experience, in contrast to DQN-based algorithms [72, 37, 7]), and is hence less efficient in harnessing extremely sparse rewards. This explanation is corroborated by the experiments done in [7], in which A3C+ (an on-policy algorithm) scores much lower than DQN (an off-policy algorithm), while using the exact same exploration bonus.

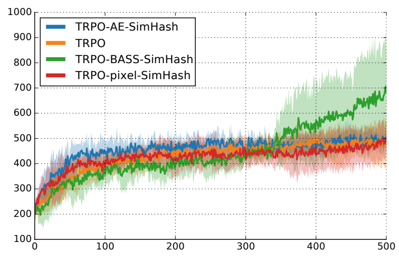
Hyperparameter Settings

Throughout all experiments, we use Adam [52] for optimizing the baseline function and the autoencoder. Hyperparameters for rllab experiments are summarized in Table 3.2. Here the policy

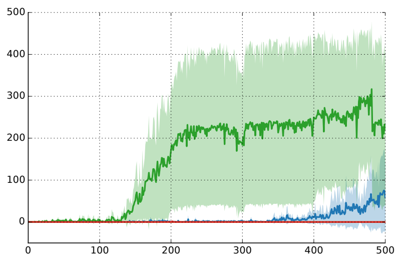
[3]We provide videos of example game play and visualizations of the difference bewteen Pixel-SimHash and BASS-SimHash at https://www.youtube.com/playlist?list=PLAd-UMX6FkBQdLNWtY8nH1-pzYJA_1T55

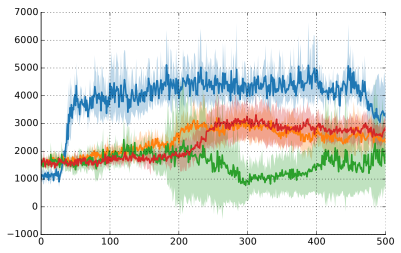

(a) Freeway


(b) Frostbite



(c) Gravitar


(d) Montezuma's Revenge


(e) Solaris

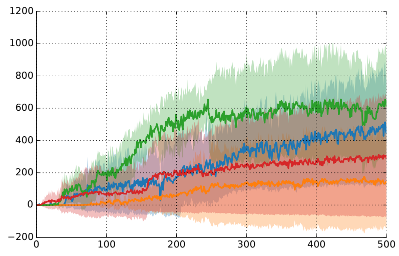

(f) Venture

Figure 3.4: Atari 2600 games: the solid line is the mean average undiscounted return per iteration, while the shaded areas represent the one standard deviation, over 5 seeds for the baseline, TRPO-pixel-SimHash, and TRPO-BASS-SimHash, while over 3 seeds for TRPO-AE-SimHash.

takes a state s as input, and outputs a Gaussian distribution $\mathcal{N}(\mu(s), \sigma^2)$, where $\mu(s)$ is the output of a multi-layer perceptron (MLP) with $\tanh$ nonlinearity, and $\sigma > 0$ is a state-independent parameter.

Table 3.2: TRPO hyperparameters for rllab experiments

Experiment	MountainCar	CartPoleSwingUp	HalfCheetah	SwimmerGatherer
TRPO batch size	5k	5k	5k	50k
TRPO step size	0.01			
Discount factor γ	0.99			
Policy hidden units	(32, 32)	(32,)	(32, 32)	(64, 32)
Baseline function	linear	linear	linear	MLP: 32 units
Exploration bonus	$\beta = 0.01$			
SimHash dimension	$k = 32$			

Hyperparameters for Atari 2600 experiments are summarized in Table 3.3 and 3.4. By default, all convolutional layers are followed by ReLU nonlinearity.

The autoencoder architecture was shown in Figure 1 of Section 2.3. Specifically, uniform noise $U(-a, a)$ with $a = 0.3$ is added to the sigmoid activations. The loss function Eq.(3) (in the main text), using $\lambda = 10$, is updated every $j_{\text{update}} = 3$ iterations. The architecture looks as follows: an input layer of size 52×52, representing the image luminance is followed by 3 consecutive 6×6

Table 3.3: TRPO hyperparameters for Atari experiments with image input

Experiment	TRPO-pixel-SimHash	TRPO-BASS-SimHash	TRPO-AE-SimHash
TRPO batch size		100k	
TRPO step size		0.01	
Discount factor		0.995	
# random seeds	5	5	3
Input preprocessing		grayscale; downsampled to 52×52; each pixel rescaled to $[-1, 1]$ 4 previous frames are concatenated to form the input state	
Policy structure		16 conv filters of size 8×8, stride 4 32 conv filters of size 4×4, stride 2 fully-connect layer with 256 units linear transform and softmax to output action probabilities (use batch normalization[42] at every layer)	
Baseline structure		(same as policy, except that the last layer is a single scalar)	
Exploration bonus		$\beta = 0.01$	
Hashing parameters	$k = 256$	cell size $C = 20$ $B = 20$ bins	$b(s)$ size: 256 bits downsampled to 64 bits

Table 3.4: TRPO hyperparameters for Atari experiments with RAM input

Experiment	TRPO-RAM-SimHash
TRPO batch size	100k
TRPO step size	0.01
Discount factor	0.995
# random seeds	10
Input preprocessing	vector of length 128 in the range $[0, 255]$; downsampled to $[-1, 1]$
Policy structure	MLP: (32, 32, number_of_actions), $\tanh$
Baseline structure	MLP: (32, 32, 1), $\tanh$
Exploration bonus	$\beta = 0.01$
SimHash dimension	$k = 256$

convolutional layers with stride 2 and 96 filters feed into a fully connected layer of size 1024, which connects to the binary code layer. This binary code layer feeds into a fully-connected layer of 1024 units, connecting to a fully-connected layer of 2400 units. This layer feeds into 3 consecutive 6×6 transposed convolutional layers of which the final one connects to a pixel-wise softmax layer with 64 bins, representing the pixel intensities. Moreover, label smoothing is applied to the different softmax bins, in which the log-probability of each of the bins is increased by 0.003, before normalizing. The softmax weights are shared among each pixel.

In addition, we apply counting Bloom filters [20] to maintain a small hash table.

3.5 A Case Study of Montezuma's Revenge

Montezuma's Revenge is widely known for its extremely sparse rewards and difficult exploration [7]. While our method does not outperform [7] on this game, we investigate the reasons behind this through various experiments. The experiment process below again demonstrates the importance of a hash function having the correct granularity and encoding relevant information for solving the MDP.

Our first attempt is to use game RAM states instead of image observations as inputs to the policy, which leads to a game score of 2500 with TRPO-BASS-SimHash. Our second attempt is to manually design a hash function that incorporates domain knowledge, called *SmartHash*, which uses an integer-valued vector consisting of the agent's (x, y) location, room number and other useful RAM information as the hash code. The best SmartHash agent is able to obtain a score of 3500. Still the performance is not optimal. We observe that a slight change in the agent's coordinates does not always result in a semantically distinct state, and thus the hash code may remain unchanged. Therefore we choose grid size s and replace the x coordinate by $\lfloor (x - x_{\min})/s \rfloor$ (similarly for y). The bonus coefficient is chosen as $\beta = 0.01\sqrt{s}$ to maintain the scale relative to the true reward[4] (see Table 3.5). Finally, the best agent is able to obtain 6600 total rewards after training for 1000 iterations ($1000\,\mathrm{M}$ time steps), with a grid size $s = 10$.

Table 3.6 lists the semantic interpretation of certain RAM entries in Montezuma's Revenge. SmartHash, as described in Section 3.5, makes use of RAM indices 3, 42, 43, 27, and 67. "Beam walls" are deadly barriers that occur periodically in some rooms.

During our pursuit, we had another interesting discovery that the ideal hash function should not simply cluster states by their visual similarity, but instead by their relevance to solving the MDP. We experimented with including enemy locations in the first two rooms into SmartHash ($s = 10$), and observed that average score dropped to 1672 (at iteration 1000). Though it is important for the agent to dodge enemies, the agent also erroneously "enjoys" watching enemy motions at distance (since new states are constantly observed) and "forgets" that his main objective is to enter other rooms. An alternative hash function keeps the same entry "enemy locations", but instead only puts randomly sampled values in it, which surprisingly achieves better performance (3112). However, by ignoring enemy locations altogether, the agent achieves a much higher score (5661) (see Figure 3.5). In retrospect, we examine the hash codes generated by BASS-SimHash and find that codes clearly distinguish between visually different states (including various enemy locations), but fails to emphasize that the agent needs to explore different rooms. Again this example showcases the importance of encoding relevant information in designing hash functions.

3.6 Discussion

The proposed hashing techniques generalize count-based exploration to more complex state spaces. However, no theoretical guarantee applies to such non-tabular state representations, unless further

[4]The bonus scaling is chosen by assuming all states are visited uniformly and the average bonus reward should remain the same for any grid size.

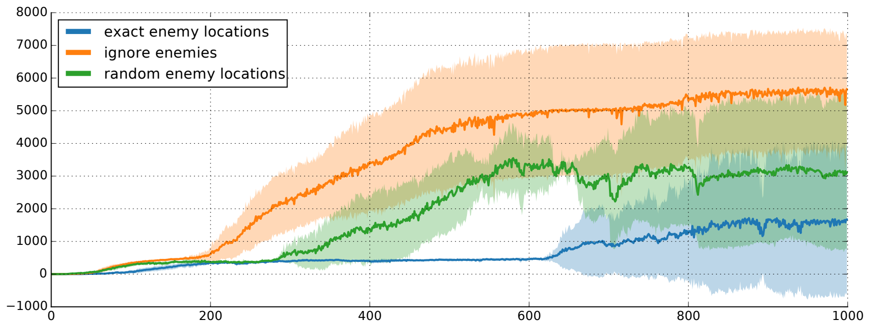


Figure 3.5: SmartHash results on Montezuma's Revenge (RAM observations): the solid line is the mean average undiscounted return per iteration, while the shaded areas represent the one standard deviation, over 5 seeds.

Table 3.5: Average score at $50\,\mathrm{M}$ time steps achieved by TRPO-SmartHash on Montezuma's Revenge (RAM observations)

s	1	5	10	20	40	60
score	2598	2500	**3533**	3025	2500	1921

Table 3.6: Interpretation of particular RAM entries in Montezuma's Revenge

ID	Group	Meaning
3	room	room number
42	agent	x coordinate
43	agent	y coordinate
52	agent	orientation (left/right)
27	beams	on/off
83	beams	beam countdown (on: 0, off: $36 \to 0$)
0	counter	counts from 0 to 255 and repeats
55	counter	death scene countdown
67	objects	Doors, skull, and key in 1st room
47	skull	x coordinate (1st and 2nd room)

assumptions about the state space structure are made and are consistent with the chosen hash function. It becomes clearer with evidence from Section 3.5 that the best hash functions should encode task semantics. Thus the user can utilize his prior knowledge about the task to accelerate convergence to the global optimum.

Though count-based exploration shapes the reward, it is essentially still solving the original problem, since all bonus rewards will converge to zero eventually. In Chapter 4, we will employ various techniques, including hierarchy, to reformulate the problem and to dramatically speed up exploration.

Chapter 4

Modular Architecture for StarCraft II with Deep Reinforcement Learning

4.1 Introduction

Deep reinforcement learning (deep RL) has become a promising tool for acquiring competitive game-playing agents, achieving success on Atari [67], Go [99], Minecraft [110], Dota 2 [79], and many other games. It is capable of processing complex sensory inputs, leveraging massive training data, and bootstrapping performance without human knowledge via self-play [100]. However, StarCraft II, a well recognized new milestone for AI research, continues to present a grand challenge to deep RL due to its complex visual input, large action space, imperfect information, and long horizon. In fact, the direct end-to-end learning approach cannot even defeat the easiest built-in AI [121].

StarCraft II is a real-time strategy game that involves collecting resources, building production facilities, researching technologies, and managing armies to defeat the opponent. Its predecessor StarCraft has attracted numerous research efforts, including hierarchical planning [124] and tree search [117] (see survey by Ontañón et al. [76]). Most prior approaches focus on substantial manual designs, yet still unable to defeat professional players, potentially due to their inability to utilize game play experiences [51].

We believe that deep RL with properly integrated human knowledge can effectively reduce the complexity of the problem without compromising policy expressiveness or performance. To achieve this goal, we propose a flexible modular architecture that shares the decision responsibilities among multiple independent modules, including worker management, build order, tactics, micro-management, and scouting (Figure 4.1). Each module can be manually scripted or handled by a neural network policy, depending on whether the task is routine and hence easy to handcraft, or highly complex and requires learning from data. All modules suggest macros (predefined action sequences) to the scheduler, which decides their order of execution. In addition, an updater keeps track of environment information and adaptively executes macros selected by the scheduler.

We further evaluate the modular architecture by reinforcement learning with self-play, focusing

Module	Responsibility	Current Design
Worker management	Ensure that resources are gathered at maximum efficiency	Scripted
Build order	Choose what unit/building/upgrade to produce	FC policy
Tactics	Choose where to send the army (attack or retreat)	FCN policy
Micromanagement	Micro-manage units to destroy more opposing units	Scripted
Scouting	Send scouts and track opponent information	Scripted + LSTM prediction

Table 4.1: The responsibility of each module and its design in our current version. FC = fully connected network. FCN = fully convolutional network.

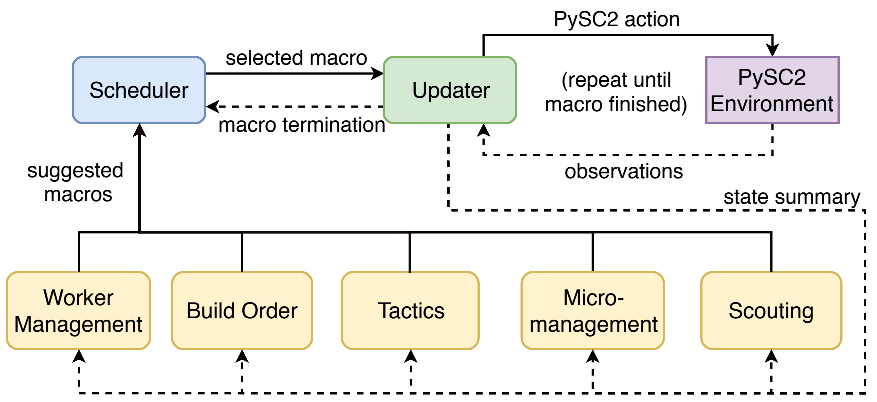


Figure 4.1: The proposed modular architecture for StarCraft II

on important aspects of the game that can benefit more from massive training experiences, including build order and tactics. The agent is trained on the PySC2 environment [121] that exposes a challenging human-like control interface. We adopt an iterative training approach that first trains one module while others follow very simple scripted behaviors, and then replace the scripted component of another module with a neural network policy, which continues to train while the previously trained modules remain fixed. We evaluate our agent playing Zerg v.s. Zerg against built-in bots on ladder maps, obtaining win rates 92% or 86% against the "Harder" bot, with or without fog-of-war. Furthermore, our agent generalizes well to held-out test maps and achieves similar performance.

Our main contribution is to demonstrate that deep RL and self-play combined with the modular architecture and proper human knowledge can achieve competitive performance on StarCraft II. Though this chapter focuses on StarCraft II, it is possible to generalize the presented techniques to other complex problems that are beyond the reach of the current end-to-end RL training paradigm.

4.2 Related Work

Classical approaches towards playing the full game of StarCraft are usually based on planning or search. Notable examples include case-based reasoning [3], goal-driven autonomy [124], and Monte-Carlo tree search [117]. Most other research efforts focus on a specific aspect of the decision hierarchy, namely strategy (macromanagement), tactics, and reactive control (micromanaging). See the survey of Ontañón et al. [76] and Robertson and Watson [89] for complete summaries. Our modular architecture is inspired by hierarchical and modular designs that won past AIIDE competitions, especially UAlbertaBot [14], but features the integration of deep RL training and self-play instead of intense hard-coding.

Reinforcement learning studies how to act optimally in a Markov Decision Process to maximize the discounted sum of rewards $R = \sum_{t=0}^{T} \gamma^t r_t$ $(\gamma \in (0, 1])$. The book by Sutton and Barto [109] gives a good overview. Deep reinforcement learning uses neural networks to represent the policy and/or the value function, which can approximate arbitrary functions and process complex inputs (e.g. visual information).

Recently, Vinyals et al. [121] have released PySC2, a python interface for StarCraft II AI, and evaluated state-of-the-art deep RL methods. Their end-to-end training approach, although shows potential for integrating deep RL to RTS games, cannot beat the easiest built-in AI. Other efforts of applying deep learning or deep RL to StarCraft (I/II) include controlling multiple units in micromanagement scenarios [86, 23, 118, 97] and learning build orders from human replays [45]. To our knowledge, no published deep RL approach has succeeded in playing the full game yet.

Optimizing different modules can also be cast as a cooperative multi-agent learning problem. Apart from aforementioned multi-agent learning works on micromanagement, other promising methods include optimistic and hysteretic Q learning [57, 64, 75], and centralized critic with decentralized actors [63]. Here we use a simple iterative training approach that alternately optimizes a single module while keeping others fixed, though incorporating multi-agent learning methods is possible and can be future work.

Self-play is a powerful technique to bootstrap from an initially random agent, without access to external data or existing agents. The combination of deep learning, planning, and self-play led to the well-known Go-playing agents AlphaGo [99] and AlphaZero [100]. More recently, Bansal et al. [5] has extended self-play to asymmetric environments and learns complex behavior of simulated robots.

4.3 Modular Architecture

Table 4.1 summarizes the role and design of each module. In the following sections, we will describe them in details for our implemented agent playing the Zerg race . Note that the design presented here is only an instance of all possible ways to implement this modular architecture. One can incorporate other methods, such as planning, into one of the modules as long as it works coherently with other modules.

Updater

The updater serves as a memory unit, a communication hub for modules, and a portal to the PySC2 environment.

Module	Macro name and inputs	Executed sequence of macros or PySC2 actions
(All)	*jump_to_base* (base)	move_camera (base.minimap_location)
	select_all_bases	select_control_group (bases_hotkey)
Worker	*rally_workers* (base)	(1) *jump_to_base* (base), (2) *select_all_bases*, (3) rally_workers_screen (base.minerals_screen_location)
	inject_larva	(1) select_control_group (Queens_hotkey), (2) for each base: (2.1) *jump_to_base* (base), (2.2) effect_inject_larva_screen (base.screen_location)
Build order	*hatch* (unit_type)	(choose by unit_type, e.g. Zergling $\rightarrow$ train_zergling_quick)
	hatch_multiple_units (unit_type, n)	(1) *select_all_bases*, (2) select_larva, (3) *hatch* (unit_type) n times
	build_new_base	(1) base = closest unoccupied base (informed by the updater) (2) *jump_to_base* (base), (3) select_any_worker, (4) build_hatchery_screen (base.screen_location)
Tactics	*attack_location* (minimap_location)	(1) select_army, (2) attack_minimap (minimap_location)
Micros	*burrow_lurkers*	(1) select_army, (2) select_unit (lurker), (3) burrowdown_lurker_quick
Scouting	*send_scout* (minimap_location)	(1) select_overlord, (2) move_minimap (minimap_location)

Table 4.2: Example macros available to each module. A macro (italic) consists of a sequence of macros or PySC2 actions (non-italic). Information such as base.minimap_location, base.screen_location and bases_hotkey is provided by the updater.

Name	Description
Time	Total time (seconds) passed
Friendly bases	Minimap locations and worker counts
Enemy bases	Minimap locations (scouted)
Neutral bases	Minimap locations
Friendly units	Friendly unit types and counts
Enemy units	Enemy unit types and counts (scouted)
Buildings	All constructed building types
Upgrades	All researched upgrades
Build queue	Units and buildings in production
Notifications	Any message from or to modules

Table 4.3: Examples memories maintained by the updater

To allow a fair comparison between AI and humans, Vinyals et al. [121] define observation inputs from PySC2 as similar to those exposed to human players, including imagery feature maps of the camera screen and the minimap (e.g. unit type, player identity), and a list of non-spatial features such as the total amount of minerals collected. Because past actions, past events, and out-of-camera information are crucial for decision making but not directly accessible from current observations, the agent has to develop an efficient memory. Though it is possible to learn such a memory from experiences, we think a properly hand-designed set of memories can serve a similar purpose, while also reducing the burden on reinforcement learning. Table 4.3 lists example memories the updater maintains. Some memories (e.g. build queue) can be inferred from previous actions taken. Some (e.g. friendly units) can be inferred from inspecting the list of all units. Others (e.g. enemy units) require further processing PySC2 observations and collaborating with the scouting module.

The “notifications” entry holds any information a module wants to notify other modules, thus allowing them to communicate and cooperate. For example, when the build order module decides to build a new base, it notifies the tactics module, which may move armies to protect the new base.

Finally, the updater handles communication between the agent and PySC2 by concretizing macros into sequences of PySC2 actions and executing them in the environment.

Macros

When playing StarCraft II, humans usually choose their actions from a list of subroutines, rather than from raw environment actions. For example, to build a new base, a player identifies an unoccupied neutral base, selects a worker, and then builds a base there. Here we name these subroutines as *macros* (examples shown in Table 4.2). Learning a policy to output macros directly can hide the trivial execution details of certain higher level commands, therefore allowing the policy to explore different strategies more effectively.

Build Order

A StarCraft II agent must balance our consumption of resources between many needs, including supply (population capacity), economy, combat units, upgrades, etc. The build order module plays the crucial role of choosing the correct thing to build. For example, in the early game, the agent needs to focus on building enough workers to gather resources, and while in the mid game, it should choose the correct types of armies that can beat the opposing ones. Though there exist numerous efficient build orders developed by professional players, executing one naively without adaptation can result in highly exploitable behavior. Instead of relying on complex if-else logic or planning to handle various scenarios, the agent’s build order module can benefit effectively from massive game-play experiences. Therefore we choose to optimize this module by deep reinforcement learning.

Here we start with a classic hardcoded build order [1], as the builds are often the same towards the beginning of the game, and the optimal trajectory is simple but requires precisely timed commands. Once the hard-coded build is exhausted, a neural network policy takes control (See Figure 4.2). This policy operates once every 5 seconds. Its input consists of the agent’s resources (minerals, gas, larva, and supply), its building counts, its unit counts, and enemy unit counts (assuming no fog-of-war). We choose to exclude spatial inputs like screen and minimap features, because choosing what to build is a high-level strategic choice that depends more on the global information. The output is the type of unit or structure to produce. For units (Drones, Overlords, and combat units), it also chooses an amount $n \in \{1, 2, 4, 8, 16\}$ to build. For structures (Hatchery, Extractor) or Queen, it only produces one at a time. The policy uses a fully connected (FC) network with four hidden layers and 512 hidden units for each layer.

We also mask out invalid actions, such as producing more units than the current resources can afford, to enable efficient exploration. If a unit type requires a certain tech structure (e.g. Roaches

[1]Exactly the first 2 minutes taken from
https://lotv.spawningtool.com/build/56414/

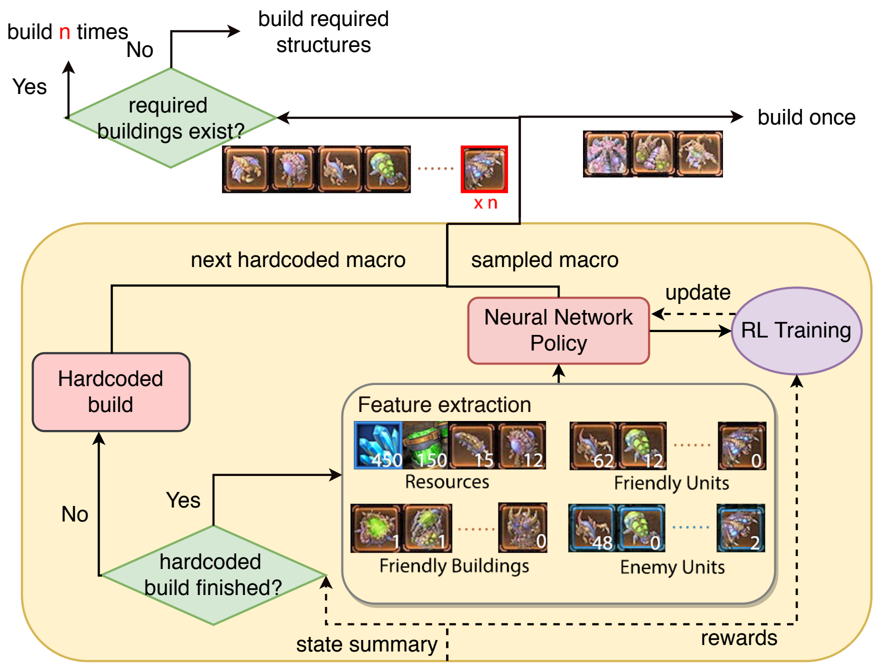


Figure 4.2: Details of our build order module.

need a Roach Warren) but it doesn't exist and is not under construction, then the policy will build the tech structure instead.

Tactics

Once our agent possesses an army provided by the build order module, it must learn to use it effectively. The tactics module handles map-level army commands, such as attacking or retreating to a specific location with a group of units. Though it is possible to hardcode certain tactics, we will show in the Evaluation section that a learned tactics can perform better.

In the current version, our tactics simply decides where on the minimap to move all of its armies towards. The input consists of 64×64 bitmaps of friendly units, enemy units (assuming no fog-of-war), and all selected friendly units on the minimap. The output is a probability distribution over minimap locations. The policy uses a three-layer Fully Convolutional Network (FCN) with 16, 32 and 1 filters, 5, 3 and 1 kernel sizes, and 2, 1 and 1 strides respectively. A softmax operation over the FCN output gives the probability over the minimap. The advantage of FCN is that its output is invariant to translations in the input, allowing the agent to generalize better to new scenarios or even new maps. The learned tactics policy operates every 10 seconds.

Scouting

Because the fog-of-war hides certain areas, enemy-dependent decisions can be very difficult to make, such as building the correct army types to counter the opponent's.

Our current agent assumes no fog-of-war during self-play training, but can be evaluated under fog-of-war at test time, with a scouting module that supplies missing information. In particular, the scouting module sends Overlords to several predefined locations on the map, regularly moves the camera to those places and updates enemy information. It maintains an exponential moving average estimate of enemy unit counts for the build order module, and uses a neural network to predict enemy unit locations on the minimap for the tactics module. The prediction neural network applies two convolutions with 16, 32 filters and 5, 3 kernel sizes to the current minimap, followed by an LSTM of 512 hidden units, whose output is reshaped to the same size of the minimap, followed by pixel-wise sigmoid to predict the probabilities of enemy units. Future work will involve adding further predictions beyond enemy locations and using RL to manage scouts.

Micromanagement

Micromanagement requires issuing precise commands to individual units, such as attacking specific opposing units, in order to maximize combat outcomes. Here we use simple scripted micros in our current version, leaving the learning of more complex behavior to future work, for example, by leveraging existing techniques presented in the Related Work. Our current micromanagement module operates when the updater detects that friendly units are close to enemies. It moves the camera to the combat location, groups up the army, attacks the location with most enemies nearby, and uses a few special abilities (e.g. burrowing lurkers, spawning infested terrans).

Worker Management

Worker management has been extensively studied for StarCraft: Brood War [13]. StarCraft II simplifies the process, so the suggested worker assignment (2 per mineral patch, 3 per vespene geyser) is often sufficient for professional players. We script this module by letting it constantly review worker counts of all bases and transferring excess workers to under-saturated mining locations, prioritizing gas over minerals. It also periodically commands Queens to inject larva at all bases, which is crucial for boosting unit production.

Scheduler

The PySC2 environment places a restriction on the number of actions per minute (APM) to ensure a fair comparison between AI and human. Therefore when multiple modules propose too many macros at the same time, not all macros can be executed and a scheduler is required to order them by priority. Our current version uses little APM, so the scheduler simply cycles through all modules and executes the oldest macro proposals. When APM increases in the future, for example when complex micromanagement comes into play, a cleverer or even learned scheduler will be required.

4.4 Training Procedure

Our agent is trained to play Zerg v.s. Zerg on the ladder map Abyssal Reef on the 4.0 version of StarCraft II. For most games, Fog-of-war is disabled. Note that built-in bots can also utilize full observations, so the comparison is fair. Each game lasts 60 minutes, after which a tie is declared.

Self-Play

We follow the self-play procedure suggested by Bansal et al. [5] to save snapshots of the current agent into a training pool periodically (every 3×10^6 policy steps). Each game the agent plays against a random opponent sampled uniformly from the training pool. To increase the diversity of opponents, we initialize the training pool with a random agent and other scripted modular agents that use fixed build orders and simple scripted tactics. The fixed build orders are optimized[2] to prioritize specific unique types and include *zerglings*, *banelings*, *roaches_and_ravagers*, *hydralisks*, *mutalisks*, *roaches_and_infestors*, and *corruptors_and_broodlords*. Zerglings are available to every build order. The scripted tactics attacks the enemy bases with all armies whenever its army supply is above 50 or its total supply is above 100. The agent never faces the built-in bots until test time.

Reinforcement Learning

If winning games is the only concern, in principle the agent should only receive a binary win-loss reward. However, we have found that the win-loss provides too sparse training signals and thus slows down training. Instead we use the supply difference (d_t) between the agent and the enemy as a reward function. A positive supply difference is often correlated with an advantageous status. Specifically, to ensure that the game is always zero-sum, the reward is the *change* in supply difference $r_t = d_t - d_{t-1}$ for each time step. Summing up all rewards yields a total reward equal to the end-game supply difference.

We use Asynchronous Advantage Actor-Critic [65] to optimize the policies with 18 parallel CPU workers. The learning rate is 10^{-4} and the entropy bonus coefficient is 10^{-1} for build order, 10^{-4} for tactics (smaller due to a larger action space). Each worker commits a gradient update to the central parameter server every 3 minutes in game (every 40 gradient steps for build order and 20 for tactics)

Iterative Training

One major benefit of the modular architecture is that modules act relatively independently and can therefore be optimized separately. We illustrate this by comparing iterative training, namely optimizing one module while keeping others fixed, against joint training, namely optimizing all modules together. We hypothesize that iterative training can be more effective because it stabilizes the experiences gathered by the training module and avoids the complex module-wise coordination during joint training.

[2]Most builds taken from https://lotv.spawningtool.com/build/zvz/

In particular, we pretrain a build order module with a scripted tactics described in the Self Play section, and meanwhile pretrain a tactics module with a scripted build order (all Roaches). Once both pretrained modules stabilize, we combine them, freeze the tactics, and only train the build order. After build order stabilizes, we freeze its parameters and train tactics instead. The procedure is abbreviated "iterative, pretrained build order and tactics".

4.5 Evaluation

Videos our agent playing against itself and qualitative analysis of the tactics module are available on https://sites.google.com/view/modular-sc2-deeprl. In this section, we would like to analyze the quantitative and qualitative performance of our agent by answering the following questions.

1. Does our agent trained with self-play outperform scripted modular agents and built-in bots?
2. Does iterative training outperform joint training?
3. How does the learned build order behave qualitatively? E.g. does it choose army types that beat the opponent's?
4. Can our agent generalize to other maps after being trained on only one map?

Quantitative Performance

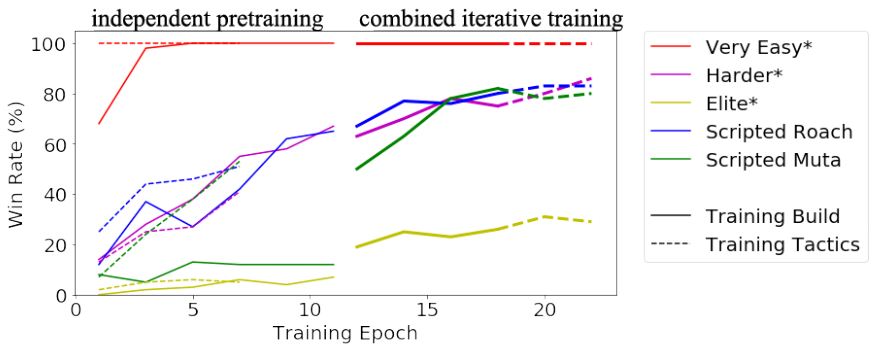


Figure 4.3: Win rates of our agent against opponents of different strengths. Asterisks indicate built-in bots that are not seen during training. 1 epoch = 3×10^5 policy steps.

Figure 4.3 shows the win rates of our agent throughout training, under the "iterative, pretrained build order and tactics" procedure. Pretrained build order and tactics can already achieve 67% and 41% win rates against the Harder bot. The win rate of combined modules increase to 86% after iterative training. Moreover, it outperforms simple scripted agents (also see Table 4.5), indicating the effectiveness of reinforcement learning.

Training procedure	Hard	Harder	Very Hard	Elite
Iterative, no pretrain	77±6 %	62±10%	11±7%	11±5%
Iterative, pretrained tactics	84±4 %	73±8%	16±5%	9±6%
Iterative, pretrained build order	**85±7%**	81±5%	39±9%	25±8%
Iterative, pretrained build order and tactics	84±5%	**87±3%**	**57±7%**	**31±11%**
Joint, no pretrain	41±21%	21±13%	4±4%	5±4%
Joint, pretrained build order	65±18%	35±9%	10±4%	6±3%

Table 4.4: Comparison of final win rates between different training procedures against built-in bots (3 seeds, averaged over 100 matches per seed)

Table 4.4 shows that iterative training outperforms joint training by a large margin. Even when joint training also starts with a pretrained bulid order, its stability quickly drops and results in 50% less win rate than iterative against the Harder bot. Pretraining both build order and tactics lead to better overall performance.

Qualitative Evaluation of the Learned Build Order

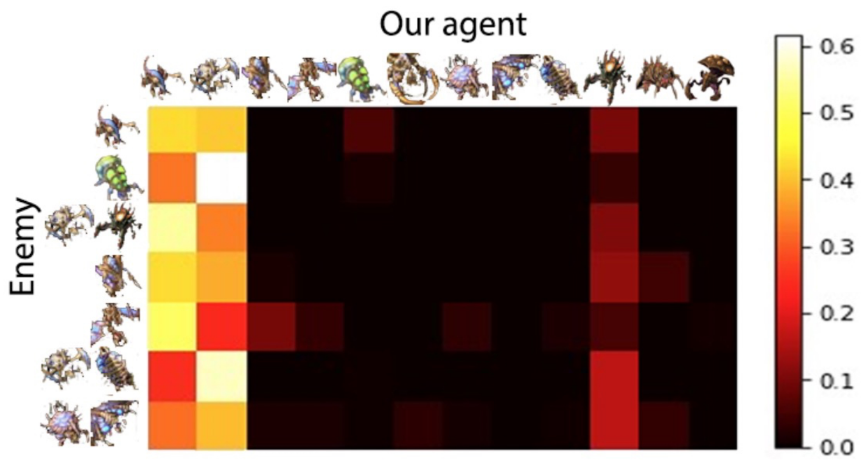


Figure 4.4: Learned army compositions. Showing ratios of total productions of each unit type to the total number of produced combat units.

Figure 4.4 shows how our learned build order module reacts to different scripted opponents that it sees during training. Though our agent prefers the Zergling-Roach-Ravager composition in general, it can correctly react to the Zerglings with more Banelings, to Banelings with fewer Zerglings and more Roaches, and to Mutalisks with more hydralisks. Currently, the reactions are not perfectly tailored to the specific opponents, likely because the Zergling-Roach-Ravager composition is strong enough to defeat the opponents before they can produce enough units.

	Average	Roach	Mutalisk	V. Easy	Easy	Medium	Hard	Harder	V. Hard	Elite
Modular (None)	67±2%	69±3%	41±6%	100%	92±2%	71±4%	77±3%	62±9%	11±4%	11±5%
Modular (Tactics)	63±4%	76±6%	29±5%	100%	100%	91±2%	84±3%	73±7%	16±6%	9±4%
Modular (Build)	76±3%	81±3%	43±6%	100%	96±1%	92±1%	**85±2%**	81±3%	39±7%	25±5%
Modular (Both)	**83±2%**	80±7%	**67±5%**	**100%**	**100%**	**99%**	84±3%	**87±2%**	**57±5%**	**31±10%**
Scripted Roaches	62±3%	–	18±9%	100%	100%	92±3%	71±7%	35±4%	6±3%	5 ±2%
Scripted Mutalisks	71±2%	**82±5%**	–	100%	99%	86±4%	77±5%	64±7%	15±4%	2 ±1%

Table 4.5: Comparison of win rates (out of 100 matches) against various opponents. Pretrained component in parenthesis. "V." means "Very".

Generalization to Different Maps

Many parts of the agent, including the modular architecture, macros, choice of policy inputs, and neural network architectures (specifically FCN), are designed with certain prior knowledge that can help with generalization to different scenarios. We test the effect of prior knowledge by evaluating our agent against different opponents on maps not seen during training. These test maps have various sizes, terrains, and mining locations. The 4-player map Darkness Sanctuary even randomly spawns players at 2 out or 4 locations. Table 4.6 summarizes the results. Though our agent's win rates drop by 6% on average against Harder, it is still very competitive.

Opponent	AR	DS	AP
Scripted Roaches	80±7%	82±4%	78±11%
Hard	84±3%	84±6%	78±5%
Harder	87±2%	77±7%	82±6%
Very Hard	57±5 %	44±7%	55±4%
Elite	31±10 %	22±11%	30±10%

Table 4.6: Win rates (out of 100 matches) of our agent against different opponents on various maps. Our agent is only trained on AR. AR = Abyssal Reef. DS = Darkness Sanctuary. AP = Acid Plant.

Opponent	Hard	Harder	V. Hard	Elite
Win Rate	95±1%	94±2%	50±8%	60±8%

Table 4.7: Win rates (out of 100 matches) of our agent on Abyssal Reef with fog-of-war enabled

Testing under Fog-of-War

Though the agent was trained without fog-of-war, we can test its performance by filling missing information with estimates from the scouting module. Table 4.7 shows that the agent actually performs much better under fog-of-war, achieving 9.5% higher win rates on average, potentially because the learned build orders and tactics generalize better to noisy/imperfect information, while the built-in agents rely on concrete observations.

4.6 Discussion

Here we have presented a suite of techniques to simplify exploration in a very challenging task. Notably, we use shaped dense rewards (Section 4.4), macro actions (Table 4.2) or hierarchical learning, a modular architecture (Section 4.3), and pretraining plus iterative finetuning (Section 4.4). These altogether accelerate learning dramatically. For example, our agent uses a total of 2 years game play experiences, while the famous AlphaStar project [120] uses 200 years for each agent, and hundreds of agents in the population. If computation resources are available, the data-intensive approach of AlphaStar can achieve better performance more likely, because it maintains the original problem. However, in resource-constrained industrial applications, it can be more favorable to speed up exploration by using techniques presented here.

Chapter 5

Conclusion

In this thesis, we have investigated three out of six categories (Section 1.4) of techniques for exploration in deep RL: entropy regularization, novelty bonus, and hierarchy. The entropy bonus idea resulted in the soft Q-learning algorithm. Questions still remained regarding how to quickly and accurately train the energy-based policy, and how to unbiasedly sample state-action pairs for soft Q-learning. It was later extended to Soft Actor-Critic [34] and had wider application. However, the core assumption that entropy bonus helps exploration, is still unchallenged. Is it possible that another information-theoretic quantity gives the same elegant theories, but also allows task-specific customization? Addressing this question can potentially strengthen existing algorithms and guide entropy bonus towards more informed exploration.

The novelty bonus, represented by hash exploration in the thesis, has been studied very frequently [10]. Despite its empirical success, the very definition of novelty has been vaguely stated and varies between papers. In fact, most works propose a universal novelty form and measure it on all tasks, but very few consider adapting it to each problem. The most effective bonus should be both novel and task-oriented, as we have seen in Section 3.5. Investigation along this direction is still lacking, but it has greater potential impact on practical applications.

Hierarchical RL has been long considered useful in exploration, due to its temporally extended reasoning. Chapter 4 presented a handcrafted hierarchy, but designing it requires significant domain knowledge. Moreover, a badly designed hierarchy can even limit the agent's capacity, compared to a non-hierarchical one[71]. Therefore it is important to acquire better hierarchy, either via learning [119, 25, 71] or a meta-optimization procedure (e.g. running evolution algorithms on the goal space). Existing works still lack systematic studies of what hierarchy is learned and what the ideal hierarchy should be. Research along this direction can help reduce the difficulty of training hierarchical agents and make existing algorithms more applicable.

We have seen that exploration informed by prior knowledge can be very powerful compared to generic methods. However, due to the *ad hoc* nature of informed exploration techniques, their most successful stories are only on isolated benchmark problems like Montezuma's Revenge [91], Dota [78], and StarCraft II [120]. For deep RL techniques to be feasible in practical problems with human and computational resource constraints, such as industrial applications, we still need more systematic study of informed exploration methods.

The last category, meta-exploration or in general meta reinforcement learning, can potentially allow fast adaptation to specific RL problems by transferring knowledge from similar tasks [17, 21]. It can offload or complement some *ad hoc* designs mentioned above, by only requiring a variety of similar problems and a neural network architecture that can handle the complex learning dynamics. It will be exciting to see more research efforts in this direction in the future.

Bibliography

[1] David Abel et al. “Exploratory Gradient Boosting for Reinforcement Learning in Complex Domains”. In: *arXiv preprint arXiv:1603.04119* (2016).

[2] Joshua Achiam and S. Shankar Sastry. “Surprise-Based Intrinsic Motivation for Deep Reinforcement Learning”. In: *ArXiv* abs/1703.01732 (2017).

[3] David W Aha, Matthew Molineaux, and Marc Ponsen. “Learning to Win: Case-Based Plan Selection in A Real-Time Strategy Game”. In: *International Conference on Case-Based Reasoning*. Springer. 2005, pp. 5–20.

[4] Alexandr Andoni and Piotr Indyk. “Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions”. In: *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 2006, pp. 459–468.

[5] Trapit Bansal et al. “Emergent Complexity via Multi-Agent Competition”. In: *International Conference on Learning Representations* (2018).

[6] Marc G Bellemare et al. “The arcade learning environment: An evaluation platform for general agents”. In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279.

[7] Marc G Bellemare et al. “Unifying Count-Based Exploration and Intrinsic Motivation”. In: *Advances in Neural Information Processing Systems 29 (NIPS)*. 2016, pp. 1471–1479.

[8] Ronen I Brafman and Moshe Tennenholtz. “R-max-a general polynomial time algorithm for near-optimal reinforcement learning”. In: *Journal of Machine Learning Research* 3 (2002), pp. 213–231.

[9] Yuri Burda et al. “Exploration by Random Network Distillation”. In: *ArXiv* abs/1810.12894 (2019).

[10] Yuri Burda et al. “Large-Scale Study of Curiosity-Driven Learning”. In: *ArXiv* (2019).

[11] Moses S Charikar. “Similarity estimation techniques from rounding algorithms”. In: *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*. 2002, pp. 380–388.

[12] Richard Y. Chen et al. “UCB and InfoGain Exploration via Q-Ensembles”. In: *ArXiv* abs/1706.01502 (2017).

[13] Dion Bak Christensen et al. *Efficient Resource Management in StarCraft: Brood War*. `https://projekter.aau.dk/projekter/files/42685711/report.pdf`. 2010.

[14] David Churchill. *UAlbertaBot*. https://github.com/davechurchill/ualbertabot. 2017.

[15] Navneet Dalal and Bill Triggs. "Histograms of oriented gradients for human detection". In: *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2005, pp. 886–893.

[16] C. Daniel, G. Neumann, and J. Peters. "Hierarchical Relative Entropy Policy Search." In: *AISTATS*. 2012, pp. 273–281.

[17] Yan Duan et al. "RL2: Fast Reinforcement Learning via Slow Reinforcement Learning". In: *ArXiv* abs/1611.02779 (2017).

[18] Y. Duan et al. "Benchmarking deep reinforcement learning for continuous control". In: *Int. Conf. on Machine Learning*. 2016.

[19] S. Elfwing et al. "Free-energy based reinforcement learning for vision-based navigation with high-dimensional sensory inputs". In: *Int. Conf. on Neural Information Processing*. Springer. 2010, pp. 215–222.

[20] Li Fan et al. "Summary cache: A scalable wide-area web cache sharing protocol". In: *IEEE/ACM Transactions on Networking* 8.3 (2000), pp. 281–293.

[21] Chelsea Finn, Pieter Abbeel, and Sergey Levine. "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks". In: *International Conference on Machine Learning*. 2017, pp. 1126–1135.

[22] C. Florensa, Y. Duan, and Abbeel P. "Stochastic Neural Networks for Hierarchical Reinforcement Learning". In: *Int. Conf. on Learning Representations*. 2017.

[23] Jakob Foerster et al. "Counterfactual Multi-Agent Policy Gradients". In: *arXiv preprint arXiv:1705.08926* (2017).

[24] R. Fox, A. Pakman, and N. Tishby. "Taming the noise in reinforcement learning via soft updates". In: *Conf. on Uncertainty in Artificial Intelligence*. 2016.

[25] Kevin Frans et al. "Meta learning shared hierarchies". In: *arXiv preprint arXiv:1710.09767* (2017).

[26] Justin Fu, John D. Co-Reyes, and Sergey Levine. "EX2: Exploration with Exemplar Models for Deep Reinforcement Learning". In: *NIPS*. 2017.

[27] Mohammad Ghavamzadeh et al. "Bayesian Reinforcement Learning: A Survey". In: *Foundations and Trends in Machine Learning* 8.5-6 (2015), pp. 359–483.

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. "Deep Learning". In: `http://www.deeplearningbook.org`. MIT Press, 2016.

[29] Karol Gregor et al. "Towards Conceptual Compression". In: *Advances in Neural Information Processing Systems 29 (NIPS)*. 2016, pp. 3549–3557.

[30] S. Gu et al. “Q-Prop: Sample-Efficient Policy Gradient with An Off-Policy Critic”. In: *arXiv preprint arXiv:1611.02247* (2016).

[31] Arthur Guez et al. “Bayes-Adaptive Simulation-based Search with Value Function Approximation”. In: *Advances in Neural Information Processing Systems (Advances in Neural Information Processing Systems (NIPS))*. 2014, pp. 451–459.

[32] Abhishek K. Gupta et al. “Meta-Reinforcement Learning of Structured Exploration Strategies”. In: *ArXiv* abs/1802.07245 (2018).

[33] Tuomas Haarnoja et al. “Reinforcement learning with deep energy-based policies”. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 1352–1361.

[34] Tuomas Haarnoja et al. “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor”. In: *arXiv preprint arXiv:1801.01290* (2018).

[35] R. Hafner and M. Riedmiller. “Reinforcement learning in feedback control”. In: *Machine Learning* 84.1-2 (2011), pp. 137–169.

[36] Hado van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-Learning”. In: *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI)*. 2016.

[37] Hado van Hasselt et al. “Learning functions across many orders of magnitudes”. In: *arXiv preprint arXiv:1602.07714* (2016).

[38] Kaiming He et al. “Deep residual learning for image recognition”. In: 2015.

[39] N. Heess, D. Silver, and Y. W. Teh. “Actor-Critic Reinforcement Learning with Energy-Based Policies”. In: *Workshop on Reinforcement Learning*. Citeseer. 2012, p. 43.

[40] N. Heess et al. “Learning and Transfer of Modulated Locomotor Controllers”. In: *arXiv preprint arXiv:1610.05182* (2016).

[41] Rein Houthooft et al. “VIME: Variational Information Maximizing Exploration”. In: *Advances in Neural Information Processing Systems 29 (NIPS)*. 2016, pp. 1109–1117.

[42] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. 2015, pp. 448–456.

[43] M. Jaderberg et al. “Reinforcement learning with unsupervised auxiliary tasks”. In: *arXiv preprint arXiv:1611.05397* (2016).

[44] Thomas Jaksch, Ronald Ortner, and Peter Auer. “Near-optimal regret bounds for reinforcement learning”. In: *Journal of Machine Learning Research* 11 (2010), pp. 1563–1600.

[45] Niels Justesen and Sebastian Risi. “Learning Macromanagement in StarCraft from Replays using Deep Learning”. In: *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*. IEEE. 2017, pp. 162–169.

[46] L. P. Kaelbling, M. L. Littman, and A. W. Moore. “Reinforcement learning: A survey”. In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[47] S. Kakade. “A natural policy gradient”. In: *Advances in Neural Information Processing Systems* 2 (2002), pp. 1531–1538.

[48] H. J. Kappen. “Path integrals and symmetry breaking for optimal control theory”. In: *Journal of Statistical Mechanics: Theory And Experiment* 2005.11 (2005), P11011.

[49] Michael Kearns and Satinder Singh. “Near-optimal reinforcement learning in polynomial time”. In: *Machine Learning* 49.2-3 (2002), pp. 209–232.

[50] T. Kim and Y. Bengio. “Deep directed generative models with energy-based probability estimation”. In: *arXiv preprint arXiv:1606.03439* (2016).

[51] Yoochul Kim and Minhyung Lee. “Intelligent Machines Humans Are Still Better Than AI at StarCraft—for Now”. In: *MIT Technology Review* (2017). URL: `https://www.technologyreview.com/s/609242/humans-are-still-better-than-ai-at-starcraftfor-now/`.

[52] D. Kingma and J. Ba. “Adam: A method for stochastic optimization”. In: 2015.

[53] J Zico Kolter and Andrew Y Ng. “Near-Bayesian exploration in polynomial time”. In: *Proceedings of the 26th International Conference on Machine Learning (ICML)*. 2009, pp. 513–520.

[54] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet classification with deep convolutional neural networks”. In: *Advances in Neural Information Processing Systems 25 (NIPS)*. 2012, pp. 1097–1105.

[55] Tejas D. Kulkarni et al. “Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation”. In: *NIPS*. 2016.

[56] T. L. Lai and H. Robbins. “Asymptotically efficient adaptive allocation rules”. In: *Advances in Applied Mathematics* 6.1 (1985), pp. 4–22.

[57] Martin Lauer and Martin Riedmiller. “An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems”. In: *In Proceedings of the Seventeenth International Conference on Machine Learning*. Citeseer. 2000.

[58] S. Levine and P. Abbeel. “Learning neural network policies with guided policy search under unknown dynamics”. In: *Advances in Neural Information Processing Systems*. 2014, pp. 1071–1079.

[59] S. Levine et al. “End-to-end training of deep visuomotor policies”. In: *Journal of Machine Learning Research* 17.39 (2016), pp. 1–40.

[60] T. P. Lillicrap et al. “Continuous control with deep reinforcement learning”. In: *ICLR* (2016).

[61] Q. Liu and D. Wang. “Stein variational gradient descent: A general purpose Bayesian inference algorithm”. In: *Advances In Neural Information Processing Systems*. 2016, pp. 2370–2378.

[62] David G Lowe. “Object recognition from local scale-invariant features”. In: *Proceedings of the 7th IEEE International Conference on Computer Vision (ICCV)*. 1999, pp. 1150–1157.

[63] Ryan Lowe et al. “Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 6382–6393.

[64] Laëtitia Matignon, Guillaume J Laurent, and Nadine Le Fort-Piat. “Hysteretic Q-Learning: An Algorithm for Decentralized Reinforcement Learning in Cooperative Multi-Agent Teams”. In: *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*. IEEE. 2007, pp. 64–69.

[65] Volodymyr Mnih et al. “Asynchronous Methods for Deep Reinforcement Learning”. In: *International Conference on Machine Learning*. 2016, pp. 1928–1937.

[66] Volodymyr Mnih et al. “Asynchronous methods for deep reinforcement learning”. In: *arXiv preprint arXiv:1602.01783* (2016).

[67] Volodymyr Mnih et al. “Human-level Control through Deep Reinforcement Learning”. In: *Nature* 518.7540 (2015), p. 529.

[68] V. Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), pp. 529–533.

[69] V. Mnih et al. “Playing atari with deep reinforcement learning”. In: *arXiv preprint* (2013).

[70] Ofir Nachum et al. “Data-Efficient Hierarchical Reinforcement Learning”. In: *NeurIPS*. 2018.

[71] Ofir Nachum et al. “Near-optimal representation learning for hierarchical reinforcement learning”. In: *arXiv preprint arXiv:1810.01257* (2018).

[72] Arun Nair et al. “Massively parallel methods for deep reinforcement learning”. In: *arXiv preprint arXiv:1507.04296* (2015).

[73] G. Neumann. “Variational inference for policy search in changing situations”. In: *Int. Conf. on Machine Learning*. 2011, pp. 817–824.

[74] B. O’Donoghue et al. “PGQ: Combining policy gradient and Q-learning”. In: *arXiv preprint arXiv:1611.01626* (2016).

[75] Shayegan Omidshafiei et al. “Deep Decentralized Multi-task Multi-Agent Reinforcement Learning under Partial Observability”. In: *International Conference on Machine Learning*. 2017, pp. 2681–2690.

[76] Santiago Ontañón et al. “A Survey of Real-Time Strategy Game AI Research and Competition in Starcraft”. In: *IEEE Transactions on Computational Intelligence and AI in games* 5.4 (2013), pp. 293–311.

[77] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. “Pixel recurrent neural networks”. In: *Proceedings of the 33rd International Conference on Machine Learning (ICML)*. 2016, pp. 1747–1756.

[78] OpenAI. *OpenAI Five*. `https://blog.openai.com/openai-five/`. 2018.

[79] OpenAI. *OpenAI Five, 2018.* `https://blog.openai.com/openai-five/`. Accessed: 2018-08-19. 2018.

[80] Ian Osband, Benjamin Van Roy, and Zheng Wen. "Generalization and Exploration via Randomized Value Functions". In: *Proceedings of the 33rd International Conference on Machine Learning (ICML)*. 2016, pp. 2377–2386.

[81] Ian Osband et al. "Deep Exploration via Bootstrapped DQN". In: *Advances in Neural Information Processing Systems 29 (NIPS)*. 2016, pp. 4026–4034.

[82] M. Otsuka, J. Yoshimoto, and K. Doya. "Free-energy-based reinforcement learning in a partially observable environment." In: *ESANN*. 2010.

[83] Pierre-Yves Oudeyer and Frederic Kaplan. "What is intrinsic motivation? A typology of computational approaches". In: *Frontiers in Neurorobotics* 1 (2007), p. 6.

[84] Deepak Pathak et al. "Curiosity-Driven Exploration by Self-Supervised Prediction". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)* (2017), pp. 488–489.

[85] Jason Pazis and Ronald Parr. "PAC Optimal Exploration in Continuous Space Markov Decision Processes". In: *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI)*. 2013.

[86] Peng Peng et al. "Multiagent Bidirectionally-Coordinated Nets for Learning to Play StarCraft Combat Games". In: *arXiv preprint arXiv:1703.10069* (2017).

[87] J. Peters, K. Mülling, and Y. Altun. "Relative Entropy Policy Search." In: *AAAI Conf. on Artificial Intelligence*. 2010, pp. 1607–1612.

[88] K. Rawlik, M. Toussaint, and S. Vijayakumar. "On stochastic optimal control and reinforcement learning by approximate inference". In: *Proceedings of Robotics: Science and Systems VIII* (2012).

[89] Glen Robertson and Ian Watson. "A Review of Real-Time Strategy Game AI". In: *AI Magazine* 35.4 (2014), pp. 75–104.

[90] Ruslan Salakhutdinov and Geoffrey Hinton. "Semantic hashing". In: *International Journal of Approximate Reasoning* 50.7 (2009), pp. 969–978.

[91] Tim Salimans and Richard Chen. "Learning Montezuma's Revenge from a Single Demonstration". In: *ArXiv* abs/1812.03381 (2018).

[92] B. Sallans and G. E. Hinton. "Reinforcement learning with factored states and actions". In: *Journal of Machine Learning Research* 5.Aug (2004), pp. 1063–1088.

[93] Jürgen Schmidhuber. "Formal theory of creativity, fun, and intrinsic motivation (1990–2010)". In: *IEEE Transactions on Autonomous Mental Development* 2.3 (2010), pp. 230–247.

[94] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint* (2017).

[95] J. Schulman et al. “High-dimensional continuous control using generalized advantage estimation”. In: *arXiv preprint arXiv:1506.02438* (2015).

[96] J. Schulman et al. “Trust Region Policy Optimization.” In: *Int. Conf on Machine Learning*. 2015, pp. 1889–1897.

[97] Kun Shao, Yuanheng Zhu, and Dongbin Zhao. “StarCraft Micromanagement with Reinforcement Learning and Curriculum Transfer Learning”. In: *IEEE Transactions on Emerging Topics in Computational Intelligence* (2018).

[98] E. Shelhamer et al. “Loss is its own Reward: Self-Supervision for Reinforcement Learning”. In: *arXiv preprint arXiv:1612.07307* (2016).

[99] David Silver et al. “Mastering the Game of Go with Deep Neural Networks and Tree Search”. In: *Nature* 529.7587 (2016), pp. 484–489.

[100] David Silver et al. “Mastering the Game of Go without Human Knowledge”. In: *Nature* 550.7676 (2017), p. 354.

[101] D. Silver et al. “Deterministic Policy Gradient Algorithms”. In: *Int. Conf on Machine Learning*. 2014.

[102] D. Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587 (Jan. 2016). Article, pp. 484–489. ISSN: 0028-0836.

[103] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[104] Bradly C Stadie, Sergey Levine, and Pieter Abbeel. “Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models”. In: *arXiv preprint arXiv:1507.00814* (2015).

[105] Alexander L Strehl and Michael L Littman. “A theoretical analysis of model-based interval estimation”. In: *Proceedings of the 21st International Conference on Machine Learning (ICML)*. 2005, pp. 856–863.

[106] Alexander L Strehl and Michael L Littman. “An analysis of model-based interval estimation for Markov decision processes”. In: *Journal of Computer and System Sciences* 74.8 (2008), pp. 1309–1331.

[107] Yi Sun, Faustino Gomez, and Jürgen Schmidhuber. “Planning to be surprised: Optimal Bayesian exploration in dynamic environments”. In: *Proceedings of the 4th International Conference on Artificial General Intelligence (AGI)*. 2011, pp. 41–51.

[108] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[109] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[110] Chen Tessler et al. “A Deep Hierarchical Approach to Lifelong Learning in Minecraft.” In: *AAAI*. Vol. 3. 2017, p. 6.

[111] P. Thomas. “Bias in Natural Actor-Critic Algorithms.” In: *Int. Conf. on Machine Learning*. 2014, pp. 441–448.

[112] E. Todorov. “General duality between optimal control and estimation”. In: *IEEE Conf. on Decision and Control*. IEEE. 2008, pp. 4286–4292.

[113] E. Todorov. “Linearly-solvable Markov decision problems”. In: *Advances in Neural Information Processing Systems*. MIT Press, 2007, pp. 1369–1376.

[114] Engin Tola, Vincent Lepetit, and Pascal Fua. “DAISY: An efficient dense descriptor applied to wide-baseline stereo”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.5 (2010), pp. 815–830.

[115] M. Toussaint. “Robot trajectory optimization using approximate inference”. In: *ICML*. ACM. 2009, pp. 1049–1056.

[116] G. E. Uhlenbeck and L. S. Ornstein. “On the theory of the Brownian motion”. In: *Physical review* 36.5 (1930), p. 823.

[117] Alberto Uriarte and Santiago Ontañón. “Improving Monte Carlo Tree Search Policies in StarCraft via Probabilistic Models Learned from Replay Data”. In: *AIIDE*. 2016.

[118] Nicolas Usunier et al. “Episodic Exploration for Deep Deterministic Policies: An Application to StarCraft Micromanagement Tasks”. In: *International Conference on Learning Representations* (2017).

[119] Alexander Vezhnevets et al. “Strategic Attentive Writer for Learning Macro-Actions”. In: *Advances in Neural Information Processing Systems 29 (NIPS)*. 2016.

[120] Oriol Vinyals et al. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`. 2019.

[121] Oriol Vinyals et al. “StarCraft II: A New Challenge for Reinforcement Learning”. In: *arXiv preprint arXiv:1708.04782* (2017).

[122] D. Wang and Q. Liu. “Learning to draw samples: With application to amortized mle for generative adversarial learning”. In: *arXiv preprint arXiv:1611.01722* (2016).

[123] Ziyu Wang, Nando de Freitas, and Marc Lanctot. “Dueling network architectures for deep reinforcement learning”. In: *Proceedings of the 33rd International Conference on Machine Learning (ICML)*. 2016, pp. 1995–2003.

[124] Ben George Weber, Michael Mateas, and Arnav Jhala. “Applying Goal-Driven Autonomy to StarCraft.” In: *AIIDE*. 2010.

[125] Ronald J Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[126] J. Zhao, M. Mathieu, and Y. LeCun. “Energy-based generative adversarial network”. In: *arXiv preprint arXiv:1609.03126* (2016).

[127] B. D. Ziebart. “Modeling purposeful adaptive behavior with the principle of maximum causal entropy”. PhD thesis. 2010.

[128] B. D. Ziebart et al. “Maximum Entropy Inverse Reinforcement Learning”. In: *AAAI Conference on Artificial Intelligence*. 2008, pp. 1433–1438.