Improving Deep Deterministic Policy Gradient for Sparse Reward and Goal-Conditioned Continuous Control

by

Ehsan Futuhi

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

We propose an improved version of deep deterministic policy gradient (DDPG) for sparse reward and goal-conditioned reinforcement learning. To enhance exploration, we introduce *ϵt-greedy*, which uses search to generate exploratory options, focusing on less-visited states. We prove that ϵt-greedy has polynomial sample complexity under mild MDP assumptions. To more efficiently use the information provided by rewarded transitions, we design a new goal-conditioned dual experience replay buffer framework *GDRB* and use *longest n-step returns*. The resulting algorithm *ETGL-DDPG* combines ***ϵt***-greedy, **G**DRB, and **L**ongest n-step with DDPG. We evaluate ETGL-DDPG on standard sparse-reward continuous tasks, which include a maze and two robotics tasks. We show that ETGL-DDPG significantly outperforms DDPG as well as other state-of-the-art methods in all environments. Further experiments show how each strategy individually enhances the performance of DDPG.

Preface

As part of my thesis, I have submitted a paper to UAI 2024, which was co-authored by Shayan Karimi, Chao Gao, and Martin Müller. Shayan Karimi contributed mostly to the theoretical analysis in that work, while Chao Gao provided valuable advice on the method and also assisted with the writing process.

To my partner and my family

For always being by my side through all circumstances.

Imagination is more important than knowledge.

– Albert Einstein, 1929.

Acknowledgements

I want to express my gratitude to my supervisor, Martin Müller, for giving me the opportunity to work with him. Whenever I encountered an issue or felt overwhelmed by a problem, he patiently guided me through it. His words were always calming and motivating and taught me valuable lessons on how to maintain a positive and patient attitude toward both research and life. I cannot thank him enough for his invaluable support and guidance through all stages of my studies.

I would like to express my gratitude to Chao for providing valuable comments on my writing and methodology, as well as to Shayan for his significant contributions to the theoretical analysis. I also want to acknowledge the funding provided by Huawei Edmonton, without which I could not have completed my thesis. Additionally, I would like to thank my thesis committee members, Nathan Sturtevant and Levi Lelis, for taking the time out of their busy schedules to review my work. Their guidance and feedback were invaluable in helping me improve my thesis.

Lastly, I would like to express my heartfelt gratitude to the most important people in my life. Firstly, my father, who taught me how to be a compassionate and caring human being; my mother, who selflessly sacrificed her own happiness to support and encourage me in following my dreams; my sisters, who have always been a constant source of kindness and support; and last but not least, my partner, who has stood by me through thick and thin, even from a distance, and has helped me overcome all the difficulties, shortcomings, and sorrows, enabling me to continue on my journey towards achieving my goals.

Contents

1 Introduction **1**

2 Background **4**
2.1 Markov Decision Process 4
2.2 Actor-critic Algorithms 5
2.3 Options 6
2.4 Experience Replay Buffer 6
2.5 Deep Deterministic Policy Gradient (DDPG) 7

3 Related Work **8**
3.1 Exploration Methods 8
3.2 Reward Propagation 13
3.2.1 Experience Transitions 13
3.2.2 Experience Replay Buffer Design 15

4 Improving Strategies for DDPG in Goal-conditioned Tasks **17**
4.1 ϵt-Greedy: Exploration with Search 17
4.2 Building Trees in Real Time 22
4.3 A Comparison of ϵt-Greedy and Other Search Methods 23
4.4 GDRB: Goal-conditioned Dual Replay Buffer 24
4.5 Using Longest n-step Return 25
4.6 ETGL-DDPG 26

5 Experiments **29**
5.1 Environments & Experiments Setup 29
5.2 Overall Performance of ETGL-DDPG 31
5.3 Environment Coverage through Exploration 33
5.4 Effectiveness of Each New Component in ETGL-DDPG 36
5.5 The Computational Perspective 37
5.6 Distribution of Terminal States 39

6 Conclusion **44**

References **45**

Appendix **49**

List of Tables

4.1 A detailed view of assumptions for each ϵt-greedy version. . . 24

5.1 Implementation details for SAC. 31
5.2 Implementation details for ETGL-DDPG. 32
5.3 Implementation details for DDPG variants. 32
5.4 Environment details. 32
5.5 Density estimator settings 33
5.6 Analysis of the impact of budget N on environment coverage. . 37

List of Figures

4.1 (a): ϵt-greedy exploration strategy. The agent creates a tree from the current state s_t with ϵ probability. Otherwise, it uses its policy to determine the next action $a_t \sim \pi$. If the newly added node s_x to the tree is located in an unvisited area $\phi(s_x = 0)$, the path from the root to that node is returned as option O. The tree helps in avoiding obstacles and staying away from highly-visited regions (middle red area). (b): GDRB and the longest n-step return for Q-value updates. τ_1 reaches the goal (a successful episode), and τ_2 is truncated by time limit (an unsuccessful episode). The first buffer D_β stores both trajectories but D_e only stores successful trajectories. The target Q-value for state s_t is shown for both trajectories below the figure. 18

4.2 Transition groups in buffer B_M. (a) The task is defined by a pair of start and goal states. (b) The formation of transition groups in B_M after occurrence of episodes in the environment. s_{g_1} is the state of the first transition in group g_1. The state of later transitions in group g_1 must be within a distance of δ from s_{g_1}. There are two cases when a new transition is made: (s_1, a_1, s_1') belongs to an existing group g_5, while (s_2, a_2, s_2') becomes the first transition in a new group g_9 as it does not belong to existing groups. 19

5.1 All environments used in the experiments. (a) Wall-maze: the agent starts from the blue region to reach the goal in the green region. (b) U-maze: A simple robot, represented by an orange ball, navigates through a maze to reach the goal area, represented by a red ball. (c) Point-push: A simple robot shown by an orange ball must push aside two movable red blocks to reach the goal area marked by a red ball. 30

5.2 The success rates for all methods in three environments: (a) Wall-maze, (b) U-maze, (c) Point-push. The results are based on an average of 10 runs with random seeds. The shaded areas indicate one standard deviation. We train each agent for 6M frames and report the success rate at each 10^5-step checkpoint. 34

5.3 The success rates for all methods in two simplified versions of Wall-maze. (a) Wall-maze after removing some of the walls. (b) Wall-maze with a start state shifted towards the goal. (c) Success rates for all methods in Wall-maze-s1. (d) Success rates for all methods in Wall-maze-s2. 35

5.4 The environment coverage for exploration strategies in three environments: (a) Wall-maze. (b) U-maze. (c) Point-push . . 36

5.5 Analyzing the separate impact of four components on DDPG: ϵt-greedy (perfect model), ϵt-greedy (replay buffer), GRS-DRB, and longest n-step return. In Wall-maze, only ϵt-greedy versions could achieve a non-zero success rate. 38
5.6 Comparing two multi-step TD update methods: the longest n-step return, and avg8-step (average of 1 to 8 steps) 39
5.7 The agent's location at the end of episodes throughout the training in Wall-maze. 41
5.8 The agent's location at the end of episodes throughout the training in U-maze. 42
5.9 The agent's location at the end of episodes throughout the training in Point-push. 43

Chapter 1

Introduction

Goal-conditioned environments serve as an intriguing testbed for Reinforcement Learning (RL) algorithms as they present a unique challenge for agents seeking to reach a goal from a starting state. In these environments, the agent's efforts are divided into trajectories, with a time limit for the agent on each trajectory to reach its goal. If the agent fails to reach the goal, it must start a new trajectory with a new start and goal state. The agent's reward or feedback from the environment can be dense, typically in the form of Euclidean distance information, or sparse, with the agent receiving no feedback unless it reaches its goal, where it receives a positive reward. Using Euclidean distance as a negative reward may not always provide accurate feedback to the agent, especially when there are local optima in the environment that appear close to the goal, but the goal is actually not attainable from there. While it is possible to devise environment-dependent feedback, it can be challenging to define for complex and large environments [16, 18]. On the other hand, in large sparse-reward environments, it can be difficult for the agent to locate the goal if it does not receive any feedback along the trajectory.

In reinforcement learning, on-policy algorithms learn a policy from data collected by the policy itself, while off-policy algorithms use data collected by the policy itself and other policies, including recent versions of itself. Off-policy algorithms are often used in goal-conditioned tasks in robotics since they can use past experiences and experiences collected by other agents. In this work, we select the Deep Deterministic Policy Gradient (DDPG)[21] algorithm as

our baseline off-policy algorithm. To store past experiences, DDPG uses a memory called the Experience Replay Buffer [27] that contains transitions, which include the current state, action, reward, next state, and goal for each time step. The agent samples mini-batches of transitions from the buffer to update its policy. However, the buffer has limited capacity and therefore can overwrite some of its entries when it is full.

In goal-conditioned environments with sparse feedback, off-policy RL algorithms may fail to reach the goal. Matheron, Perrin, and Sigaud [23] investigated this issue in the context of DDPG and found that the agent must receive rewarded transitions early on in training; otherwise, the policy may converge to a poor state and stabilize. This unsuccessful quest to find the reward is mainly due to two reasons: First, the agent does not explore the environment effectively, resulting in a low number of successful trajectories and therefore less overall feedback. Recently, many approaches have been proposed to encourage the agent to explore the environment until they achieve a reasonable success rate in reaching the goal[6, 9, 28]. Second, it is hard for the agent to efficiently use the collected experience in the buffer. The reward resulting from a successful trajectory is only given at the final step, and the agent does not receive any positive feedback for previous steps. One solution is to share the reward among other steps with existing techniques such as n-step Temporal Difference Learning (TD) [36]. Also, since rewarded transitions are rare, they can be missed easily. A recent study [32] prioritized certain transitions over others in the buffer based on factors such as TD error. Lastly, if the agent always overwrites old experiences from the buffer, it may forget previously acquired knowledge.

In this thesis, we seek to answer the following research questions for off-policy RL algorithms in goal-conditioned environments with sparse reward:

- Can we enhance the exploration of DDPG to cover a larger portion of the environment?
- Can we maintain and sample the collected experience in a way that ensures rewarded transitions are not overlooked?

- Can we use techniques such as n-step TD in DDPG to share the sparse reward from the final step in successful trajectories with early steps?

We propose three strategies that can jointly improve the performance of DDPG in sparse goal-conditioned tasks by addressing all three research questions. Our first strategy involves implementing a simple exploration method using search trees that enables the agent to better explore the environment. Additionally, we demonstrate that this method has a polynomial sample complexity in covering state-action pairs of a discrete finite environment. Our second strategy is to create a new experience replay buffer setting that allows the agent to efficiently store and sample the collected experience for updating the policy. Lastly, our third method is to use n-step TD updates such that the achieved reward from a successful trajectory is distributed among all moves in that trajectory. This also increases the number of rewarded transitions in the buffer. We empirically demonstrate that each of these three strategies individually enhances the performance of off-policy RL. Furthermore, we show that when used in combination, they outperform current state-of-the-art methods.

Chapter 2

Background

This chapter provides background material on goal-conditioned Reinforcement Learning that is essential to understanding the rest of the thesis. In Section 2.1, we introduce the Markov decision process (MDP). In Section 2.2, we define the options framework in Reinforcement Learning. In Section 2.3, we discuss the Deep Deterministic Policy Gradients (DDPG) algorithm, which is widely used for RL in continuous state-action spaces. We use DDPG as a base algorithm for our work.

2.1 Markov Decision Process

We consider the goal-conditioned Reinforcement Learning setting where an agent interacts with an environment described as a Markov decision process (MDP), defined by the tuple $(S, A, \mathcal{T}, r, \gamma, \rho)$ where S is the set of states, A is the set of actions, $\mathcal{T}(s'|s, a)$ is the transition distribution, $r : S * A * S \to \mathbb{R}$ is the reward function, $\gamma \in [0, 1]$ is the discount factor, and ρ is the distribution from which initial and goal states are sampled for each episode. Every episode starts with sampling a new pair of initial and goal states $(s_0, s_g) \sim \rho$ where $s_0,\ s_g \in S$. At each time-step t, the agent chooses an action $a_t = \pi(s_t, s_g)$ using its policy and considering the current state and the goal state. After execution, it gets the reward $r_t = (s_t, a_t, s_g)$ and the next state sampled from $\mathcal{T}(.|s_t, a_t)$. The episode ends when the goal state or the maximum number of steps T is reached. The return is the discounted sum of future rewards $R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$. The Q-function and value function associated with the

agent's policy are defined as $Q^{\pi}(s_t, a_t, s_g) = \mathbb{E}_{\pi}[R_t|s_t, a_t, s_g]$ and $V^{\pi}(s_t, s_g) = max_a Q^{\pi}(s_t, a_t, s_g)$. The agent's objective is to learn an optimal policy π^* that maximizes the expected return $\mathbb{E}_{s_0}[R_0|s_0, s_g]$. This is often too hard in practice and good approximations are looked for. The Q-function and value function of an optimal policy are Q^* and V^*, such that $Q^*(s, a, s_g) \geq Q^{\pi}(s, a, s_g)$ and $V^*(s, s_g) \geq V^{\pi}(s, s_g)$ for every $s, s_g \in S$, $a \in A$ and any policy π.

2.2 Actor-critic Algorithms

In reinforcement learning, there are two primary types of methods: *action-value* methods and *policy gradient* methods. Action-value methods focus on learning the values of actions and then selecting the actions with the highest estimated value. Policy gradient methods aim to learn a parametrized policy $\pi(\theta)$ that can choose actions without relying on a value function. The update to $\pi(\theta)$ is based on the gradient of a scalar performance measure $J(\theta)$ with respect to θ, in order to maximize J.

$$\theta_{t+1} = \theta_t + \alpha \nabla \widehat{J(\theta_t)} \tag{2.1}$$

Here, α is the step size, and $\nabla \widehat{J(\theta_t)}$ is a stochastic estimate whose expectation approximates the gradient of J at θ_t. If the policy gradient method learns an approximate value function, then the approach is called an actor-critic method. Here, the actor is the learned policy, and the critic is the learned value function. An actor-critic algorithm with a stochastic actor $\pi_\theta : S \to P(A)$ is called a *stochastic actor-critic* algorithm [36]. Its gradients are obtained from the stochastic policy gradient theorem :

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s \sim \rho^{\pi},\ a \sim \pi_\theta}[\nabla_\theta log \pi_\theta(a|s) Q^{\pi}(s, a)], \tag{2.2}$$

where ρ^{π} is the state distribution under policy π_θ. An actor-critic algorithm with a deterministic actor $\mu_\theta : S \to A$ is called a *deterministic actor-critic* algorithm[34]. Its gradients are obtained by the deterministic policy gradient theorem:

$$\nabla_\theta J(\mu_\theta) = \mathbb{E}_{s\sim\rho^\mu}[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}], \tag{2.3}$$

where ρ^μ is the state distribution under policy μ_θ. The deterministic policy gradient is a special case of the stochastic policy gradient [34].

2.3 Options

Options [37] are traditionally defined to extend a MDP framework. An option O can be defined as a tuple $O = \langle I, \pi, \beta \rangle$, where $I \leq S$ is the initiation set of states where an option can begin, π is the option policy that determines which actions to take while executing the option, and β is the termination condition that determines when the option terminates. The reward corresponding to an option is the discounted sum of all rewards collected through the path, with discount factor γ,

$$R(s,O) = \mathbb{E}[\sum_{t'=t+1}^{t+k} \gamma^{t'-t-1} r_{t'}] \tag{2.4}$$

Here, $r_{t'}$ is the reward at time t', and the option O selected in state s at time t lasts for k steps.

2.4 Experience Replay Buffer

In off-policy RL methods, the Experience Replay Buffer [27] is used to store data collected during training. The information for time step t is stored as a transition (s_t, a_t, r_t, s_{t+1}), where s_t is a state, a_t is the agent's action, r_t is the received reward, and s_{t+1} is the next state. The agent samples a mini-batch of transitions from the buffer periodically using a *sampling policy*. Each time that the buffer is full, the *retention policy* selects some of the transitions to be replaced by new ones. The typical sampling policy is a uniform policy, where all transitions in the buffer have an equal chance of being selected. The typical retention policy is FIFO, which removes the oldest transition whenever the buffer is full. One of the crucial factors is the size of the replay buffer because if it is too small, the agent will forget previously acquired knowledge. If it is

too large, it will hinder the learning process by mainly using outdated data and selecting recent data less frequently.

2.5 Deep Deterministic Policy Gradient (DDPG)

DDPG is an actor-critic algorithm based on the deterministic policy gradient theorem (DPG) in Eq. 2.3. It is designed to work in continuous action spaces by adopting the success of Deep Q-learning [27]. Our notation uses explicit references to the goal state for both the critic and the actor networks. DDPG maintains an actor $\mu(s, s_g)$ and a critic $Q(s, a, s_g)$. The agent explores the environment through a stochastic policy $a \sim \mu(s, s_g) + w$, where w is a noise sampled from a normal distribution or an Ornstein-Uhlenbeck process [41]. To update both actor and critic, transition tuples are sampled uniformly from the replay buffer to perform a mini-batch gradient descent. The critic is updated by a loss $L = \mathbb{E}[Q(s_t, a_t, s_g) - y_t]^2$ derived from the Bellman equation,

$$Q^{\mu}(s_t, a_t) = \mathbb{E}[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1}))] \tag{2.5}$$

where $y_t = r_t + \gamma Q'(s_{t+1}, \mu'(s_{t+1}, s_g), s_g)$. Q' and μ' are the target critic and actor, respectively; their weights are soft-updated to the current weights of the main critic and actor, respectively:

$$\begin{cases} \theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'} \\ \theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'} \end{cases} \tag{2.6}$$

where τ typically is a small value such as 0.01. The actor is updated by the deterministic policy gradient algorithm [34] to maximize the estimated Q-values of the critic using loss $-\mathbb{E}_s[Q(s, \mu(s, s_g), s_g)]$.

Chapter 3

Related Work

In this chapter, we present a review of recent research on addressing the challenges posed by sparse rewards in goal-conditioned tasks. Section 3.1 focuses on methods that enhance the agent's exploration, such as using long-horizon planning algorithms, providing a reward bonus to unexplored states, or employing an additional policy solely to gather diverse samples for training the primary policy. Section 3.2 delves into techniques that effectively utilize environment rewards by shaping the reward function, prioritizing rewarded transitions, or increasing the frequency of rewarded transitions in the buffer.

3.1 Exploration Methods

In continuous action-space environments, a representative off-policy RL algorithm is deep deterministic policy gradient (DDPG) [21]. The algorithm explores by injecting noise into action decisions. However, with sparse-reward environments, DDPG does not perform well. It cannot explore enough to find rarely occurring rewarded transitions. In this section, we review some of the existing techniques that enhance exploration.

One way to improve the exploration of the agent is to use Reward Shaping [18]. As an example, assume that an agent requires a key to unlock a door in order to obtain a reward. Reward shaping could reward the agent for the intermediate goal of finding the key, in addition to opening the door. However, such techniques are specific to the environment and cannot be easily defined generally. One possible solution to shaping the reward function for

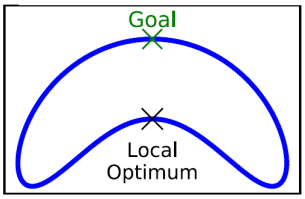


Figure 3.1: A map showing a local optimum and the main goal [40].

goal-conditioned tasks without any prior domain knowledge is *self-balancing reward shaping* [40]. This method employs the Euclidean distance as a negative reward to guide the agent toward the goal. However, an agent may get stuck in a local optimum, as shown in Figure 3.1. To overcome this issue, the self-balancing reward shaping method injects a penalty near the local optimum $\hat{g}$ into the reward function, as demonstrated in Equation 3.1,

$$r'(s,g,\hat{g}) \quad = \quad \begin{cases} 1, & d(s,g) \leq \delta \\ min[0, -d(s,g) + d(s,\hat{g})], & otherwise \end{cases} \tag{3.1}$$

where $d : S * S \rightarrow \mathbb{R}$ is the Euclidean distance function. This approach encourages the agent to simultaneously move towards the goal and away from the local optimum. Figure 3.1 depicts an environment with only one apparent local optimum $\hat{g}$. However, in many environments, it is difficult to define such a local optimum, and there may be many. Trott *et al.* [40] consider all terminal states at the end of trajectories as local optima that the agent should get away from because they are not a goal state. To this end, they sample two trajectories, known as sibling trajectories, instead of one for every episode. The terminal state of each trajectory is used as a local optimum in the reward function for the other trajectory. Self-balancing reward shaping works well in the context of on-policy learning. However, in the off-policy case, the reward function for old samples in the replay buffer is updated every time. This could result in non-stationarity, which can negatively impact performance.

Another technique to encourage the agent to explore more is to use intrinsic motivation [4, 6, 28, 30], which is receiving a reward bonus in addition to the reward from the environment for particular states. This reward bonus can act similarly to the curiosity of human agents, prompting the agent to explore

novel areas. Pathak *et al.* [30] define curiosity as the error in an agent's prediction of the consequence of its own actions in the environment. Thus, apart from learning a policy, the agent also learns the environment's model to estimate the transition dynamics. When the agent has experienced similar transitions before, the prediction error will be low, indicating a lower level of curiosity. On the other hand, if the agent has not encountered a particular transition before, the prediction error will be high, which will make the agent more curious to explore such transitions further. Burda *et al.* [6] define a reward bonus as the agent's prediction error of a fixed random target neural network instead of the environment model. This bonus is called the *Random Network Distillation (RND)* bonus. The agent has a model trained to generate similar outputs to the target network. This target network produces similar outcomes for similar inputs. Therefore, the model performs better on highly visited states as it has been trained more on them and is able to produce results that are closer to the target network.

A reward bonus can also be directly defined by tracking the number of times each state is visited. In a tabular setting, the reward bonus is inversely proportional to the visit count, as shown in Equation 3.2,

$$r'(s,a) = r(s,a) + r^{+} \quad and \quad r^{+} = \beta N(s,a)^{-\frac{1}{2}} \tag{3.2}$$

However, this approach is not effective in large state spaces, as the agent is not likely to revisit states. To overcome this limitation, Bellemare *et al.* [4] proposed using pseudo-counts, which are an approximation of visit counts derived from density models.

Intrinsic motivation can be a powerful driver for exploration, but it has some drawbacks [31]. First, assigning different rewards for a single transition breaks the Markov property and introduces non-stationarity, which can negatively impact the agent's performance. Second, hyperparameter tuning is critical to achieve good performance. Most values lead to degraded performance, and only a narrow range of values work well.

To benefit from intrinsic motivation while avoiding the aforementioned

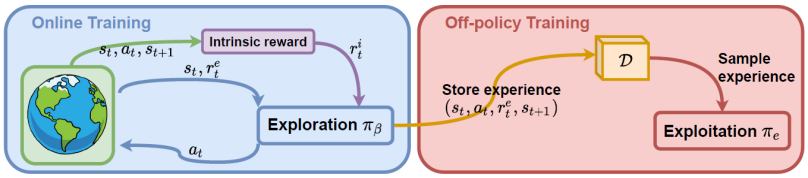


Figure 3.2: Training of decoupled RL algorithms [31].

drawbacks, decoupled reinforcement learning algorithms were introduced [3, 31]. They involve training two policies at the same time: an exploration policy π_β, and an exploitation policy π_e. The exploration policy utilizes both an intrinsic motivation r^i and the reward from environment r^e to gather a diverse set of samples from the environment. The exploitation policy is trained on data collected by the exploration policy, but the reward is relabeled to only be the environment's reward. Figure 3.2 shows the training process of decoupled RL algorithms. Both on-policy and off-policy algorithms can be used to train the exploration policy, while only off-policy algorithms are suitable for training the exploitation policy because it is trained on data from another policy. The exploitation policy is used as the main policy in evaluation and is trained at a slower rate than the exploration policy. Although the decoupled RL algorithms address the drawbacks of using intrinsic motivation, they double the computation requirements as they must train two policies.

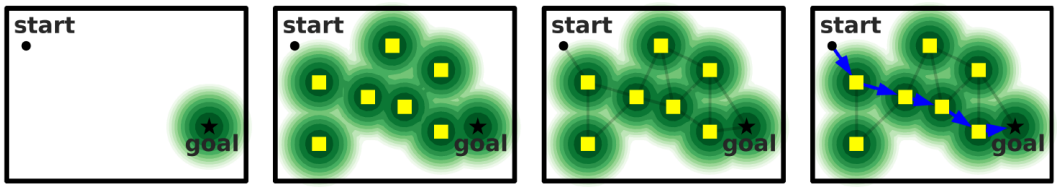
All of the previously mentioned exploration strategies were developed because simple exploration strategies such as epsilon-greedy do not work well in large, sparse-reward environments. However, Dabney, Ostrovski, and Barreto [10] claim that these strategies are environment-dependent and can be difficult to implement. Additionally, the theoretical support behind them is weak, and the exploration problem is defined as a separate problem from the main problem. On the other hand, the epsilon-greedy strategy is simple to implement, generally applicable, and has a strong theoretical grounding [35]. One of the issues with epsilon-greedy is that it lacks temporal persistence, and the agent cannot deviate far enough from the current policy distribution. To address this, Dabney, Ostrovski, and Barreto [10] propose the ϵz-greedy approach, which incorporates options into an epsilon-greedy framework. The provided

options are simple: Each option is a single primitive action repeated n times. Therefore, we have as many options as primitive actions. The option length n is sampled from a distribution whenever the option is selected. Equation 3.3 describes how the action is selected at each time step.

$$a_t = \begin{cases} \pi(s_t), & with\ probability\ 1-\epsilon \\ randomly\ select\ from\ options, & otherwise \end{cases} \tag{3.3}$$

For an option o that repeats action a, the initiation set is the state space $I = S$, the policy is $\pi(s) = a\ \forall\ s \in S$, and if the time that option selected is t_0, the termination condition is met at time $t = t_0 + n$. After option termination, the next action is selected by Eq. 3.3. The distribution generating the option length can simply be a uniform $z(n) = \mathbb{1}_{n \leq N}/N$ distribution. A heavy-tailed zeta distribution was also used in [10], denoted by $z(n) \propto n^{-\mu}$, and $\mu = 2$ showed the best performance compared to the uniform distribution.

Another exploration strategy is to utilize the long-horizon planning capabilities of search algorithms. Eysenbach, Salakhutdinov, and Levine [12] argue that off-policy RL is not ideal for tasks that require an agent to take many steps to reach a goal, but it is effective at learning a distance estimate and navigation policy for short paths. To boost the performance of off-policy RL in goal-conditioned tasks, they combine the strengths of search and goal-conditioned RL. The states in the replay buffer are viewed as a graph of the environment. A distance metric is used to determine if two nodes in the graph are connected, and Dijkstra's algorithm is used to find the shortest path between the start and goal nodes. Once the sequence of nodes in the path is found, the navigation policy is used to connect nearby nodes on the path starting from the start node. Experiments in [12] demonstrate that this *Search on the Replay Buffer* (SoRB) approach outperforms state-of-the-art methods such as HER [2] in visual navigation tasks. Figure 3.3 presents the general view of the algorithm.



(a) (b) (c) (d)

Figure 3.3: The general view of the SoRB algorithm: (a) Start and goal are added to the graph. (b) Observed states in the environment are added sequentially to the graph and the distance metric is used to determine which nodes are adjacent. (c) A pathfinding algorithm connects the nodes on the optimal path to the goal. (d) The off-policy RL algorithm enables the agent to navigate along the path between nodes [12].

3.2 Reward Propagation

In environments with sparse rewards, the agent needs to effectively utilize the information provided by rewarded transitions. To achieve a specific goal, it is important to pass on the information along successful paths to ensure that all these actions stand out. Two approaches to achieve this are to efficiently use the data stored in the replay buffer and to focus on improving the experience replay buffer setting. In Sections 3.2.1 and 3.2.2, we will review related methods.

3.2.1 Experience Transitions

As DDPG uses a one-step TD update, the reward achieved in one transition only affects states involved in the transition. To propagate the reward to other states in the trajectory, the agent must repeat the path several times. In large environments where the agent only achieves the goal a few times, reward propagation is poor. One possible remedy is to use multi-step TD methods [36]. As shown in Equation 3.4, an n-step TD target is the discounted sum of (n-1) future rewards and the Q-value of the nth state-action pair.

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(s_{t+n}, a_{t+n}) \quad (3.4)$$

In this way, we can directly use a rarely occurring reward for past transitions in the trajectory. Hessel *et al.* [15] show that using a 4-step TD update

can greatly improve the performance of DQN. Meng, Gorbet, and Kulić [24] show that using multi-step updates, especially the average of multi-step updates, can remove overestimation bias and help propagate information faster. Mnih *et al.* [25] introduce *A3C*, which uses the longest n-step return to improve the performance of actor-critic algorithms in Atari games [26]. To perform a single update, Mnih *et al.* [25] first select actions using an exploration policy for up to t_{max} steps or until a terminal state is reached. This means that a one-step update is performed for the last state, a two-step update for the second-last state, and so on, for a maximum of t_{max} updates.

Another way to improve reward propagation is to extract meaningful information from trajectories that failed to reach the goal. The agent can learn from undesired outcomes as much as from the desired ones, similar to how humans can learn from their mistakes. If the agent fails to reach the goal and ends up in an undesirable state, it can still learn about the structure of the environment. Hindsight Experience Replay (HER) [2] is one way to do so by considering *imaginary goals*. After an episode, if the agent does not reach the goal, it can consider one of the future states in the episode as an imaginary goal, which was unintentionally achieved from any earlier state in the episode. According to Equation 3.5, for each transition in an unsuccessful episode, another transition is stored in the replay buffer that treats one of the future states in the trajectory as a new imaginary goal g' with a new reward r'.

$$(s_t, a_t, r_{t+1}, s_{t+1}, g) \xrightarrow[new\ transition]{} (s_t, a_t, r'_{t+1}, s_{t+1}, g') \tag{3.5}$$

The reward r' equals 1 if the agent reaches the goal, and 0 otherwise. There are different strategies to sample imaginary goals in an episode. One of these is called *future*, which randomly samples k states from those observed after the current transition in the episode. Andrychowicz *et al.* [2] showed that off-policy RL can benefit from imaginary goals in many sparse-reward scenarios. However, it is worth noting that increasing the number of sampled imaginary goals sometimes negatively impacts performance.

3.2.2 Experience Replay Buffer Design

Improving the sampling policy is one potential enhancement to the experience replay buffer design. In environments with sparse rewards, a uniform sampling policy can often miss rewarded transitions due to their rarity in the replay buffer. To ensure that specific transitions are not missed, they can be directly included in the sampled mini-batch. In Combined Experience Replay (CER) [43], the last encountered transition in an episode is appended to the uniformly sampled mini-batch. This way, the agent's network is constantly updated with new data with minimal additional computation of $\mathcal{O}(1)$. Zhang *et al.* [43] showed that CER makes the replay buffer less sensitive to its size, resulting in good performance for a wider range of buffer sizes. However, if the buffer size is appropriately tuned, CER does not improve the performance. Another method to sample unevenly from the replay buffer is to give desired transitions more chances of being selected. Such transitions can be ones where the value estimate is far from the target value, and the agent requires more training on them than on others. Prioritized Experience Replay (PER) [32] distributes the priority among transitions according to their TD error. We can also prioritize the most rewarded transitions or the most recent ones. One difference between CER and PER to select more recent samples is that PER requires more computation $\mathcal{O}(logN)$ per sample, where N is the buffer size.

CER and PER both introduce a better sampling strategy and use a single buffer to store transitions. Zhang *et al.* [43] propose the use of two separate replay buffers, for exploitation and for exploration. The transitions made by the agent's policy are stored in the exploitation buffer, and the random exploratory transitions are stored in the exploration buffer. Although both buffers are of the same size, the ratio of samples picked from each buffer varies based on changes between the current policy network and the target network. The changes are determined by comparing the actions chosen by two networks for the same state. At the beginning of the training, the current network undergoes frequent changes, and the sampling ratio is low for the exploitation buffer and high for the exploration buffer. However, towards the end of the

training, changes become slight, and the ratios are reversed.

Zhang *et al.* [43] also replaced the FIFO retention policy with reservoir sampling [42] in order to reduce the risk of catastrophic forgetting. This is because the FIFO policy stores only the most recent transitions, which can lead to a too-local representation of the entire state space. On the other hand, with reservoir sampling all data in the replay buffer have an equal chance of being overwritten, thus maintaining more even coverage of the entire state space. In order to collect a diverse set of transitions, the exploration buffer uses reservoir sampling as the retention policy, while the exploitation buffer uses FIFO to focus on the current transitions.

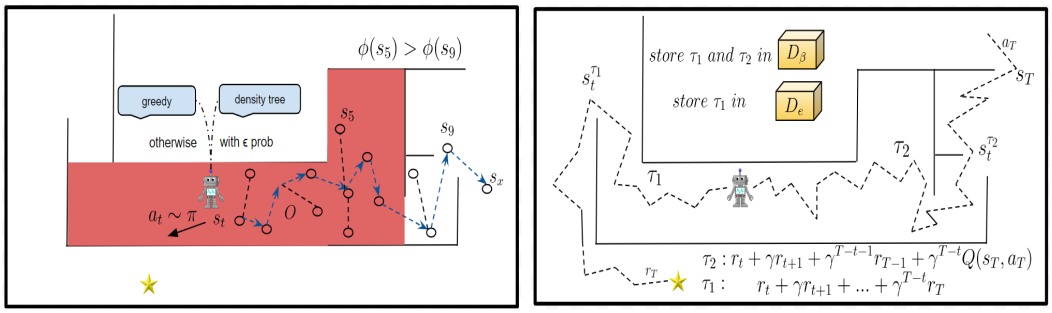
Chapter 4

Improving Strategies for DDPG in Goal-conditioned Tasks

In this chapter, we develop three strategies that can be used to improve DDPG for sparse-reward goal-conditioned tasks shown in Figure 4.1. In Section 4.1, we present a new exploration method ϵt-*greedy*, which is an extension of ϵz-greedy [10]. Section 4.2 presents a real-time version of the ϵt-greedy algorithm. The following two sections focus on ways to more efficiently use the information available from the data, as having a good exploration strategy alone is not sufficient. Section 4.4 introduces a new replay buffer *GDRB* for goal-conditioned tasks, which uses two replay buffers. Section 4.5 presents how to use n-step returns [36] to achieve better information propagation within episodes. We finally propose our new DDPG version *ETGL-DDPG* which incorporates all three strategies in Section 4.6.

4.1 ϵt-Greedy: Exploration with Search

Search algorithms have been widely used to improve the performance of RL algorithms. For example, AlphaZero [22] uses the visit distribution of root's children in the tree created by MCTS [5] to train the policy. AlphaZero assumes that the agent knows the environment model, but it can also learn a model as well, which is known as MuZero [33]. In DeepCubeA [1], A* [14] is utilized to solve the Rubik's cube by employing a heuristic learned through RL algorithms. Rapidly Exploring Random Tree (RRT) [19] is commonly used



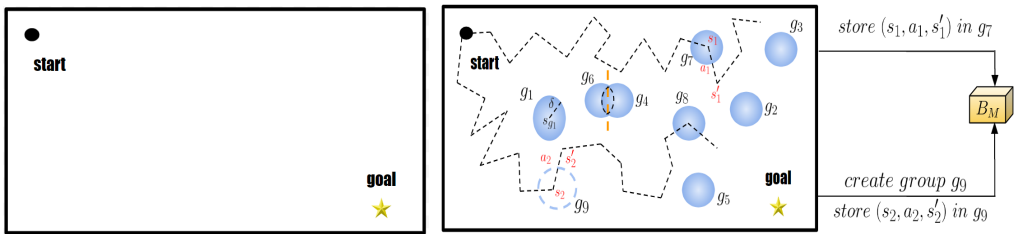
(a) ϵt-greedy: greedy or density tree (b) GDRB and the longest n-step return

Figure 4.1: (a): ϵt-greedy exploration strategy. The agent creates a tree from the current state s_t with ϵ probability. Otherwise, it uses its policy to determine the next action $a_t \sim \pi$. If the newly added node s_x to the tree is located in an unvisited area $\phi(s_x = 0)$, the path from the root to that node is returned as option O. The tree helps in avoiding obstacles and staying away from highly-visited regions (middle red area). (b): GDRB and the longest n-step return for Q-value updates. τ_1 reaches the goal (a successful episode), and τ_2 is truncated by time limit (an unsuccessful episode). The first buffer D_β stores both trajectories but D_e only stores successful trajectories. The target Q-value for state s_t is shown for both trajectories below the figure.

in navigation problems as it can explore the environment effectively and find the goal. However, RRT is not used as much in RL, such as MCTS and A*, because it requires sampling random points in the state space. Nonetheless, Chen and Müller [8] developed a method to grow the search tree similar to RRT in continuous environments without such a requirement.

Motivated by the success of the fast exploration algorithms RRT and ϵz-greedy [10], we introduce *ϵt-greedy*, by equipping ϵ-greedy with a *tree search* procedure. Like ϵ-greedy, ϵt-greedy selects a greedy action with probability $1-\epsilon$, and an exploratory action with probability ϵ. However, instead of exploring uniformly at random, ϵt-greedy chooses the first step of an *exploratory option* generated via a search with time budget N.

We provide two versions of this algorithm. In the basic version, we assume that ϵt-greedy can easily access the environment transition function $\mathcal{T}$ of the corresponding MDP. In the second version, we assume that the agent cannot search using $\mathcal{T}$; instead, it can only search upon its replay buffer. For both versions, we assume that the agent has a density function ϕ, such that for any state s, $\phi(s)$ provides an estimate of the number of visits to s during



(a) The goal-conditioned task before training.

(b) Groups are generated from the observed transitions.

Figure 4.2: Transition groups in buffer B_M. (a) The task is defined by a pair of start and goal states. (b) The formation of transition groups in B_M after occurrence of episodes in the environment. s_{g_1} is the state of the first transition in group g_1. The state of later transitions in group g_1 must be within a distance of δ from s_{g_1}. There are two cases when a new transition is made: (s_1, a_1, s_1') belongs to an existing group g_5, while (s_2, a_2, s_2') becomes the first transition in a new group g_9 as it does not belong to existing groups.

the whole learning process. When the state space is continuous and large, we approximate the visit counts by discretizing the environment into small cells, and use ϕ to estimate the visit count of these cells.

Algorithm 1 explains how the search is conducted for generating an exploratory option. Initially, at state s, we create a list of frontier nodes consisting of only the root node. We then conduct a tree search iteratively with a maximum of N iterations. At each iteration, a node s_x is sampled uniformly from the frontier nodes, and a *child* $s_{x'}$ for s_x is generated. If accessing $\mathcal{T}$ is allowed, $s_{x'}$ can be produced by randomly sampling a move from the action set $\mathcal{A}(s_x)$ (shown as `next_state_from_env` in Algorithm 1). However, accessing $\mathcal{T}$ arbitrarily breaks the *model-free* assumption of DDPG. We offer an alternative choice (shown as `next_state_from_replay_buffer`) that samples the child state from a replay buffer. If $\phi(s_{x'}) = 0$, we terminate and return the action sequence from the root to $s_{x'}$; otherwise, we repeat this process until we have added N nodes to the tree. We then return the action sequence from the root to a node s_{min} with minimum ϕ value in the tree.

The replay buffer contains transitions observed during training. It can be used as a precise transition model for observed transitions and an approximate one for similar transitions to those already seen.

To achieve this approximation, we group observed transitions in buffer B_M.

Algorithm 1 Generating exploratory option with tree search

function generate_option(state s, density estimator ϕ, budget N)
    frontier_nodes $\leftarrow$ {}
    Initialize root using s: *root* $\leftarrow$ *TreeNode*(s)
    frontier_nodes $\leftarrow$ frontier_nodes $\cup$ {root};
    $s_{\min} \leftarrow$ root
    $n \leftarrow 0$
    while $n < N$ **do**
        $s_x \sim$ *UniformRandom*(frontier_nodes)
        $s_{x'}$= **next_state_from_buffer**(s_x)
        if $\phi(s_{x'})$=0 **then**
            Extract option o by actions *root* to $s_{x'}$
            return o
        end if
        if $\phi(s_{x'}) < \phi(s_{\min})$ **then**
            $s_{\min}$=$s_{x'}$
        end if
        $n \leftarrow n + 1$
    end while
    Extract option o by actions *root* to $s_{\min}$
    return o
end function

function next_state_from_env(s_x, frontier_nodes)
    a $\sim$ *UniformRandom*($\mathcal{A}(s_x)$)
    $s_{x'} \leftarrow \mathcal{T}(s_x$, a)
    s_x.add_child ($s_{x'}$)
    frontier_nodes $\leftarrow$ frontier_nodes $\cup$ $\{s_{x'}\}$
    return $s_{x'}$
end function

function next_state_from_buffer(s_x, frontier_nodes)
    Find g as the group where s_x belongs to
    $(s', a, r, s'') \sim$ *UnifromRandom*(g)
    $s_{x'} \leftarrow s''$
    s_x.add_child($s_{x'}$)
    frontier_nodes $\leftarrow$ frontier_nodes $\cup$ $\{s_{x'}\}$
    return $s_{x'}$
end function

Buffer B_M is a set of groups, where each group comprises a set of transitions. The state of the first-joined transition to group $g \in B_M$, denoted by s_g, is used to add new transitions to g. If the agent makes a transition (s_t, a_t, r_t, s_{t+1}) in the environment and we have n groups $\{g_1, g_2, ..., g_n\} \subseteq B_M$, the transition belongs to group g_i if $d(s_t, s_{g_i}) < \delta \; for \; i \in \{1, 2, ..., n\}$. Otherwise, it is classified

as the first transition of a new group g_{n+1}. If multiple groups meet the condition, the group with the lowest d is selected, and ties are broken randomly. Here, $d : S * S \rightarrow R$ is the L^2 norm, and δ is a constant threshold. The transitions are assigned to their group in B_M upon being added to the replay buffer. Figure 4.2 depicts how observed transitions are grouped in B_M. To conduct the search upon B_M, s_x must belong to one of the groups in B_M. Assuming that s_x belongs to group g, function `next_state_from_replay_buffer` works as follows: We randomly select a transition (s', a, r, s'') in g and create a new child $s_{x'}$ for s_x by using the following approximation:

$$\mathcal{T}(s_x, a) \approx \mathcal{T}(s', a) \tag{4.1}$$

Every new node during the search is considered visited since it is chosen from an observed transition in B_M. Therefore, the building process continues until the tree budget N is reached. We then return the option from s_t to the node with minimum ϕ value.

When we use the approximation in Eq. 4.1, we assume that similar states will generate similar outcomes with the same action. This approximation works best when we have enough data in the buffer so that we can find a close neighbor for each state we encounter in the environment. We can control the precision of the approximation by adjusting δ. A large δ will create larger groups and less precision, while a small δ will yield smaller groups and greater precision.

To justify the rationale of this exploration method, we provide an analysis of the sample complexity of ϵt-greedy. Specifically, we apply the same conditions as described in [22]. We begin by defining the necessary terms and then present our main theorem. Detailed definitions and proof are presented in the Appendix.

Definition 1 (Covering Length)**.** *The covering length is the number of steps an agent needs to take starting from any* (s, a) *with* $s \in S, a \in A$*, in order to visit all state-action pairs at least once with probability at least* 0.5*.*

Definition 2 (δ-optimal policy)**.** *Policy* π *is called* δ*-optimal, if it satisfies*

$|V_{\pi^*}(s) - V_{\pi}(s)| \leq \delta$, $\forall s \in S$, $\delta > 0$.

Definition 3 (Polynomial Sample Complexity)**.** *If the number of time steps an agent needs to find a $\delta-$optimal policy, when executing algorithm A, is polynomial in all related MDP parameters, then A has a polynomial sample complexity*

Liu and Brunskill [22] showed a polynomial sample complexity for a random exploration policy by bounding the covering length. Using this concept and the fact that the tree budget N is limited, we show that ϵt-greedy also has a polynomial sample complexity.

Theorem 1. *Given state space S and action space A, and a set of options Ω_T generated by density trees which creates tree set $T = \{t_1, t_2, \dots t_k\}$, after k number of explorations. If $N \leq \Theta(|S||A|)$ and distribution over generated option denoted by $\Pr[\omega \in \Omega_T] \geq \frac{1}{\Theta(|S||A|)}$, ϵt-greedy can achieve a polynomial sample complexity.*

4.2 Building Trees in Real Time

ϵt-greedy requires access to the environment transition function to build a density tree in simulation. Through the use of the replay buffer, we were able to demonstrate that this particular requirement can be disregarded. Another solution to eliminate the need for such access is to transfer the process of building a density tree from simulation to real-time. We make Assumption 1 about the structure of the environment:

Assumption 1. *If there exists a path of actions from state A to state B, then there exists a path of actions from state B to state A.*

If we impose the above assumption on the environment, it means that all actions can be reversed. Therefore, real-time density trees are not feasible in environments like Atari games [26], where actions such as killing an enemy cannot be undone. In environments such as navigation tasks where Assumption 1 holds, a new node is added to the tree using the following steps: Since

the density tree is connected, there is always a path between any two nodes in the tree. So, at each iteration, when a new node s_x is sampled from $\mathcal{T}$, and the agent is located at another node s_c, it can traverse to s_x. The simplest way to find the path between s_c and s_x is to find the path to both of them from the root. Then, the agent returns to the root from s_c and goes to s_x. We avoid storing the agent's transitions during tree traversal in the replay buffer to prevent redundant data. When the agent reaches s_x, it executes a random action a in the environment to get a new node $s_{x'}$. It repeats this process to add nodes in the tree until one of these termination conditions is met:

- $\phi(s_{x'}) = 0$: The agent is already at $s_{x'}$, so we do not need to find the option from s_{root} to $s_{x'}$.
- budget N expires: let node s_{min} have the minimum visit count in the tree, but the agent is located at s_c. ϵt-greedy then returns the option from s_c to s_{min} as described above.

4.3 A Comparison of ϵt-Greedy and Other Search Methods

The ϵt-greedy algorithm uses search to explore the state space effectively. It grows the tree in a similar manner to RRT but does not require random point sampling in the environment. Unlike AlphaZero and DeepCubeA, where search results are used to improve neural network predictions, in ϵt-greedy, the search only offers access to unexplored areas, and learning is done through the corresponding Reinforcement Learning (RL) method. ϵt-greedy is also different from A*, which requires a heuristic function to guide the search. A* is able to detect irrelevant state-action pairs with the help of the heuristic, without exploring all possible pairs. However, in a space reward environment where the agent does not receive feedback, it must explore until it finds a promising path. Furthermore, to the best of our knowledge, there is no recent work on designing a heuristic for A* to locate unvisited states. Finally, to facilitate differentiation between the different versions of ϵt-greedy and the required

assumptions for each, we have summarized the necessary assumptions for each version in Table 4.1.

Table 4.1: A detailed view of assumptions for each ϵt-greedy version.

Algorithm	Assumptions
Using the perfect model	The transition function $\mathcal{T}$ of the environment is known.
Using the replay buffer	Similar states follow similar transitions with the same action.
Using real-time trees	If we can get to point B from point A, then we can also get to point A from point B.

4.4 GDRB: Goal-conditioned Dual Replay Buffer

The experience replay buffer is an indispensable part of deep off-policy RL algorithms. It is common to use only one buffer to store all transitions and use FIFO as the retention policy, with the most recent data replacing the oldest data [26]. With reservoir sampling [42] as the retention policy, each transition in the buffer has an equal chance of being overwritten. This maintains coverage of some older data during training. *RS-DRB* [43] uses two replay buffers, one for exploitation and the other for exploration. The transitions made by the agent's policy are stored in the exploitation buffer, and the random exploratory transitions are stored in the exploration buffer. For the retention policy, the exploration buffer uses reservoir sampling, while the exploitation buffer uses FIFO.

Inspired by the dual replay buffer framework, we propose a *Goal-conditioned Double Replay Buffer (GDRB)*. The first buffer D_β stores all transitions during training, and the second buffer D_e stores the transitions that belong to successful episodes, where the goal was reached. D_β uses reservoir sampling, and D_e uses FIFO. Since D_β needs to cover transitions from the entire training process, we make it larger than D_e. We balance the number of samples taken from the two buffers with an adaptive sampling ratio. In a training process of M episodes, at current episode n, the sampling ratios τ_e and τ_β for D_e and D_β are respectively as follows:

$$\tau_e = \frac{n}{M} \quad and \quad \tau_\beta = 1 - \tau_e \tag{4.2}$$

If C mini-batches are selected, $\max(\lfloor \tau_\beta * C \rfloor, 1)$ mini-batches are selected from D_β and the rest from D_e. Later stages of training still sample from D_β to not forget previously acquired knowledge, as we assume the policy is more likely to reach the goal as the training progresses. In case D_e is empty, there are no successful episodes yet, and we draw all mini-batches from D_β.

4.5 Using Longest n-step Return

In standard DDPG, the Q-values are updated using one-step TD. In goal-conditioned environments with sparse rewards, only one rewarded transition is added to the replay buffer per successful episode. The agent needs rewards provided by these transitions to update its policy toward reaching the goal. With few rewarded transitions, the agent should exploit a successful path to the goal many times so the reward is propagated backward quickly. Multi-step updates can accelerate this process by looking ahead several steps, resulting in more rewarded transitions in the replay buffer [15, 24]. For example, Meng, Gorbet, and Kulić [24] utilize n-step updates in DDPG with n ranging from 1 to 8. In our design, to share the reward from the last step of a successful episode for all transitions in the episode, we use *longest n-step return*, shown in Equation 4.3.

$$Q(s_t, a_t) = \begin{cases} \sum_{k=0}^{T-t} \gamma^k r_{t+k}, & s_T \text{ is a goal state} \\ \sum_{k=0}^{T-t-1} \gamma^k r_{t+k} + \gamma^{T-t} Q(s_T, a_T), & \text{otherwise} \end{cases} \tag{4.3}$$

Here, s_T is the last state in the episode. Using the longest n-step return for each transition from a successful episode, the reward is immediately propagated to all Q-value updates. In unsuccessful episodes, using the longest n-step return reduces the overestimation bias in Q-values [38]. Meng, Gorbet,

and Kulić [24] empirically show that using multi-step updates can improve the performance of DDPG on robotic tasks mostly by reducing overestimation bias — they demonstrate that the larger the number of steps, the lower the estimated target Q-value and overestimation bias.

4.6 ETGL-DDPG

This section introduces our new DDPG-based algortihm *ETGL-DDPG* which incorporates three components: $\boldsymbol{\epsilon t}$-greedy, **G**DRB, and **L**ongest n-step returns. ETGL-DDPG is shown in two sections: Algorithm 2 describes how the agent interacts with environments in episodes and Algorithm 3 shows how the collected data is used to train the networks after each episode. The parts from the original DDPG algorithm are highlighted in red in both sections.

Algorithm 2 ETGL-DDPG: Until an episode termination

Randomly initialize critic network $Q(s, a, g|\theta^Q)$ and actor $\mu(s, g|\theta^\mu)$ with weights θ^Q and θ^μ
Initialize target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffers D_β, D_e, density estimator ϕ, exploration budget N
$\epsilon \leftarrow 1$
for episodes=1,M **do**
  Receive initial observation state s_1 and goal g
  $success \leftarrow false$, $bootstrap \leftarrow 1$, $l \leftarrow 0$
  while t $\leq$ T **and not**($success$) **do**
    if l==0 **then**
      if random() $< \epsilon$ **then**
        Exploratory option $w \leftarrow$ exploration function(s_t)
        Assign action : $a_t \leftarrow w$
        $l \leftarrow$ length(w)
      else
        Greedy action : $a_t \leftarrow \mu(s_t, g|\theta^\mu)$
      end if
    else
      Assign action : $a_t \leftarrow w$
      $l \leftarrow l - 1$
    end if
    Execute action a_t and observe reward r_t and next state s_{t+1}
    if is_goal(s_{t+1}) **then**
      $success \leftarrow true$
      $bootstrap \leftarrow 0$
    end if
  end while
end for

Algorithm 3 ETGL-DDPG: After each episode termination

After each episode termination:

$$R = \begin{cases} r_t & success \\ 0 & otherwise \end{cases}$$

for $i \in \{t-1, ..., t_{start}\}$ **do**
  $R \leftarrow r_i + \gamma R$
  if *success* **then**
    store transition $(s_i, g, a_i, R, s_t, bootstrap)$ in D_β, D_e
  else
    store transition $(s_i, g, a_i, R, s_t, bootstrap)$ in D_β
  end if
end for

Sample C random mini-batches of k transitions
$(s_j, g_j, a_j, r_j, s_{j+1}, bootstrap_j)$ by τ_β and τ_e ratios from D_β and D_e
set $y_j = r_j + bootstrap_j * \gamma Q'(s_{j+1}, g_j, \mu'(s_{j+1}, g_j|\theta^{\mu'})|\theta^{Q'})$
update critic by minimizing the loss: $L = \frac{1}{k}\sum_j (y_j - Q(s_j, g_j, a_j|\theta^Q))$

update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{k} \sum_j \nabla_a Q(s, g, a|\theta^Q)|_{s=s_j, g=g_j, a=\mu(s_j, g_j)} \nabla_{\theta^\mu} \mu(s, g|\theta^\mu)|_{s_j}$$

update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu\prime} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

Chapter 5

Experiments

In this chapter, we empirically evaluate the impact of our three strategies on DDPG's performance. Our goal is to answer these questions:

1. Can ETGL-DDPG outperform state-of-the-art methods in sparse goal-conditioned tasks?

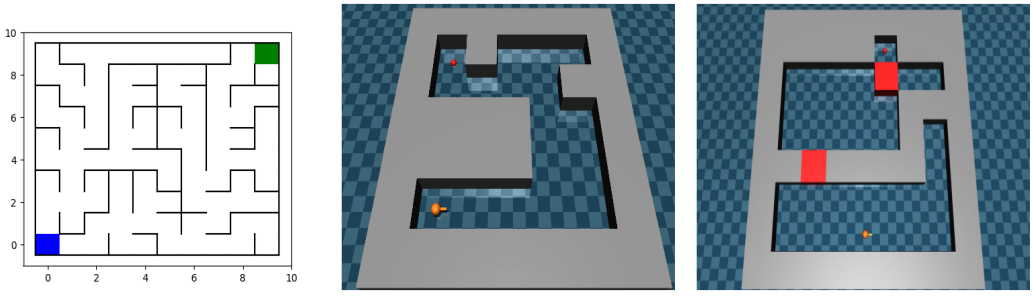
2. Can ϵt-greedy cover the environment better than ϵz-greedy and other exploration strategies?

3. Does each strategy positively impact the performance?

In the following, we introduce the test environments and describe how we designed our experiments in Section 5.1. In Section 5.2, we answer question 1. We then move on to question 2 in Section 5.3. Lastly, Section 5.4 evaluates the impact of each component individually to address question 3.

5.1 Environments & Experiments Setup

We consider three sparse-reward continuous environments. The first environment is *Wall-maze* [40]. We choose Wall-maze as it is a hard task for off-policy RL algorithms based on the experiments in [40]. In Wall-maze, a reward of -1 is given at each step, and a reward of 10 is given if the goal is reached. The start and goal states for each episode are randomly selected from the blue and green regions, respectively (Figure 5.1a). The agent's action (dx,dy) determines the amount of movement along both axes. The environment contains



(a) Wall-maze (b) U-maze (c) Ant-push

Figure 5.1: All environments used in the experiments. (a) Wall-maze: the agent starts from the blue region to reach the goal in the green region. (b) U-maze: A simple robot, represented by an orange ball, navigates through a maze to reach the goal area, represented by a red ball. (c) Point-push: A simple robot shown by an orange ball must push aside two movable red blocks to reach the goal area marked by a red ball.

a gradient cliff feature [20], where the fastest way to reach the goal results in a deadlock close to the goal. Our second and third environments are *U-maze* (Figure 5.1b) and *Point-push* (Figure 5.1c) [17], designed using the MuJoCo physics engine [39]. We add these two environments as Mujoco-designed environments are commonly used for robotics experiments. In both environments, a robot (orange ball) seeks to reach the goal (red region). In Point-push, the robot must additionally push aside the two movable red blocks that obstruct the path to the goal. A small negative reward of -0.001 is given at each step unless the goal is reached, where the reward is 1. In each episode, the robot starts near the same position with slight random variations, but the goal region remains fixed.

The maximum number of steps per episode is 100 for Wall-maze and 500 for U-maze and Point-push. We use a simple density estimator ϕ by discretizing the environment into small cells in all environments. For all baselines, we use the implementations from OpenAI Gym [11]. ϵt-greedy and ϵz-greedy use budget $N = 40$ and $N = 15$, respectively, in all environments. The budget for ϵz-greedy is smaller because its performance deteriorates with large options. The neural network structure is chosen from [40] and is the same among all methods. It has 3 hidden layers of size 128 and ReLU activation functions. All

experiments were run on a system with 5 vCPU on a cluster of Intel Xeon E5-2650 v4 2.2GHz CPUs and one 2080Ti GPU. Table 5.4 displays the details for our test environments. We present the hyperparameters to get the best results for ETGL-DDPG as well as for the baselines employing DDPG and SAC in Tables 5.2, 5.3, and 5.1. The performance of ETGL-DDPG is mainly affected by three parameters: tree budget N, ϵ decay rate, and the size of buffers D_e and D_β. On the other hand, the learning rates (for actor and critic) and the number of neural network updates have an impact on all methods. Table 5.5 provides information on the density estimator setting and the required number of visits to consider a cell *visited* for the environment coverage experiment. As the Wall-maze environment is smaller than the other two, a cell needs a higher visit count to be considered visited.

Table 5.1: Implementation details for SAC.

Hyperparameter	wall-maze	U-maze	Point-push
batch size	128		
update frequency	every 12 steps		
action noise	$\sim N(0, 0.2)$	$\sim N(0, (0.3, 0.05))$	
warmup period	$2 * 10^5$ frames		
replay buffer size	10^6		
learning rate	$3 * 10^{-4}$		
soft target updates τ	$5 * 10^{-3}$		
discount factor γ	0.99		

5.2 Overall Performance of ETGL-DDPG

We evaluate the performance of ETGL-DDPG compared to state-of-the-art methods. We compare with SAC [13], DDPG, DDPG with HER [2], DDPG with visit-counts as intrinsic motivation [28], and DDPG with ϵz-greedy [10]. The results are shown in Figure 5.2. In Wall-maze, none of the baselines achieve any success, while ETGL-DDPG, using a perfect model, gets higher success rates and finally reaches a success rate of 1. If the replay buffer is

Table 5.2: Implementation details for ETGL-DDPG.

Hyperparameter	wall-maze	U-maze	Point-push
batch size	128		
number of updates per episode	10		20
warmup period	$2 * 10^5$ frames		
exploration buffer size	$5 * 10^5$		$1 * 10^6$
exploitation buffer size	$5 * 10^4$		
actor learning rate	10^{-4}		
critic learning rate	10^{-3}		
epsilon decay rate	0.9999988		0.9999992
exploration budget N	20	40	
soft target updates τ	10^{-2}		
discount factor γ	0.99		

Table 5.3: Implementation details for DDPG variants.

Hyperparameter	wall-maze	U-maze	Point-push
batch size	128		
number of updates per episode	20		
warmup period	$2 * 10^5$ frames		
replay buffer size	10^6		
action noise	$\sim N(0, 0.2)$	$\sim N(0, (0.3, 0.05))$	
actor learning rate	10^{-4}		
critic learning rate	10^{-3}		
soft target updates τ	10^{-2}		
discount factor γ	0.99		
number of sampled goals (HER)	5	4	4
goal sampling strategy (HER)	*future*		
β (intrinsic motivation)	25		5
visit count increment (intrinsic motivation)	0.001	0.01	0.1
duration distribution (ϵz-greedy)	$z(n) = \mathbb{1}_{n \leq N}/N$		
N (ϵz-greedy)	10	15	

Table 5.4: Environment details.

environment	$S \in$	$G \in$	$A \in$	maximum number of steps per episode
Wall-maze	$\mathbb{R}^2$	$\mathbb{R}^2$	$[-0.95, 0.95]^2$	100
U-maze	$\mathbb{R}^6$	$\mathbb{R}^2$	$[-1, 1] * [-0.25, 0.25]$	500
Point-push	$\mathbb{R}^{11}$	$\mathbb{R}^2$	$[-1, 1] * [-0.25, 0.25]$	500

Table 5.5: Density estimator settings

environment	cell size	number of visits to be *visited*
Wall-maze	0.5 * 0.5	1000
U-maze	0.5 * 0.5	250
Point-push	0.5 * 0.5	200

used instead of the perfect model, the success rate stabilizes around 0.8 after 40 checkpoints. In U-maze, ETGL-DDPG converges faster than all methods except HER, although HER's success rate later deteriorates. Using the replay buffer slows the learning process, but the agent eventually achieves a success rate of 1. The use of visit counts as intrinsic motivation with DDPG performs worse than DDPG alone. This is not uncommon when one-step methods use intrinsic motivation in challenging exploration problems [28]. In Point-push, the success rate of 1 is only achieved by both versions of ETGL-DDPG. ϵz-greedy struggles as the agent's only options consist of repeating the same random action.

We also use simpler instances of Wall-maze. In *Wall-maze-s1*, there are more paths toward the goal due to removing some walls (Figure 5.3a). In *Wall-maze-s2*, the start state is shifted much closer to the goal (Figure 5.3b). All the methods are able to find the goal sometimes during training. However, in the case of DDPG, DDPG + intrinsic motivation, and DDPG + HER, the success rate remains very low. On the other hand, the performance of ϵz-greedy is highly improved as the environment has fewer obstacles between start and goal states. Both ETGL-DDPG versions still outperform all baselines. Interestingly, in Wall-maze-s2, the version using the replay buffer works slightly better than the one using the perfect model.

5.3 Environment Coverage through Exploration

We now examine how effective ϵt-greedy is in covering the environment. We discretize the environment into small cells. A cell is classified as visited if the agent visits a minimum number of different states (shown in Table 5.5)

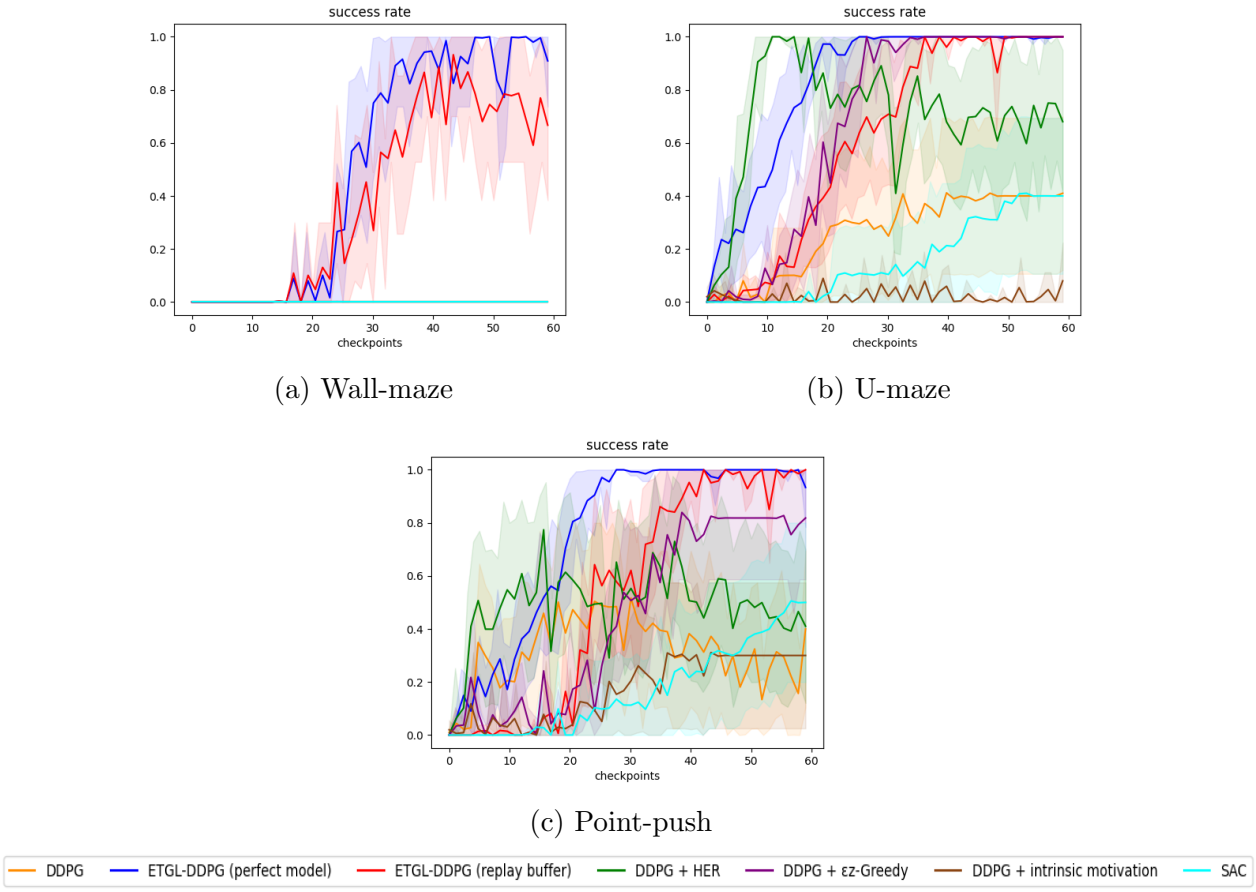
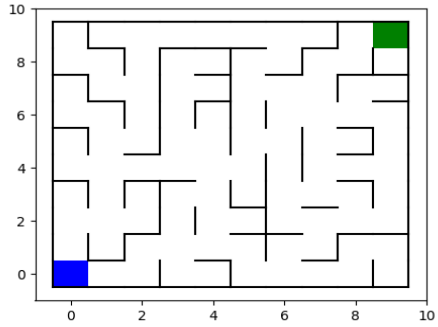
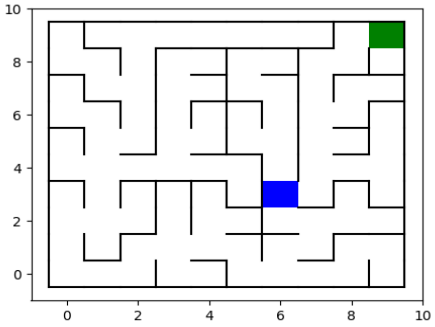


Figure 5.2: The success rates for all methods in three environments: (a) Wall-maze, (b) U-maze, (c) Point-push. The results are based on an average of 10 runs with random seeds. The shaded areas indicate one standard deviation. We train each agent for 6M frames and report the success rate at each 10^5-step checkpoint.

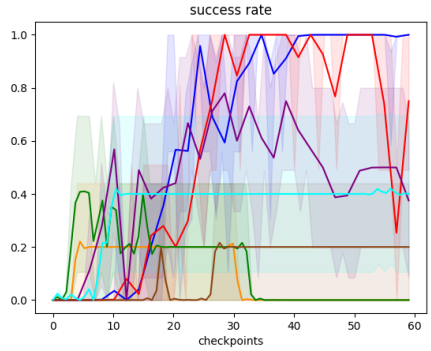
within that cell. We measure the environment coverage as the fraction of visited cells. Figure 5.4 illustrates the environment coverage for ϵt-greedy and other exploration strategies, including intrinsic motivation and ϵz-greedy. All strategies use DDPG as their underlying algorithm. In Wall-maze, only ϵt-greedy with a perfect model is capable of fully covering the environment. Using the replay buffer instead of the perfect model slightly reduces the coverage. The best baseline strategy, ϵz-greedy, covers half of the environment, while other methods can only explore about 30%. In U-maze, ϵt-greedy with a perfect model reaches full coverage faster than other methods. All tested strategies are successful, covering 80% or more of the environment. In Point-



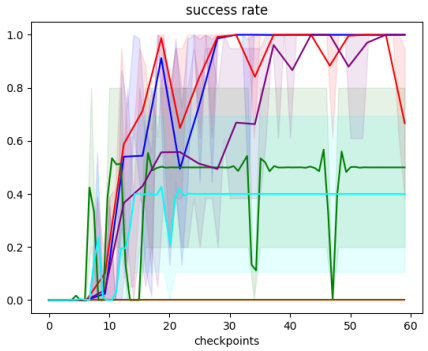
(a) Wall-maze-s1



(b) Wall-maze-s2



(c) Success rates for Wall-maze-s1

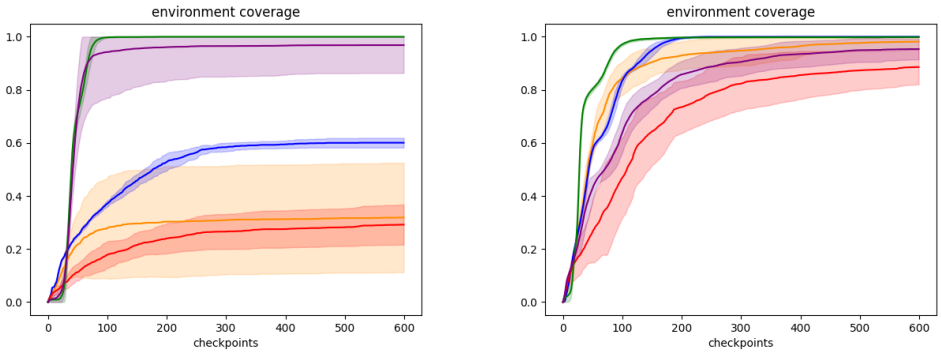


(d) Success rates for Wall-maze-s2

Figure 5.3: The success rates for all methods in two simplified versions of Wall-maze. (a) Wall-maze after removing some of the walls. (b) Wall-maze with a start state shifted towards the goal. (c) Success rates for all methods in Wall-maze-s1. (d) Success rates for all methods in Wall-maze-s2.

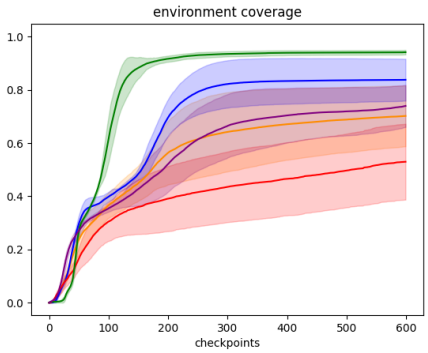
push, none of the methods can fully cover the environment. ϵt-greedy with a perfect model still outperforms the other methods. In both U-maze and Point-push, using the replay buffer slightly lowers the coverage of ϵt-greedy compared to ϵz-greedy. This is because ϵt-greedy grows the density tree by already observed transitions when using the replay buffer, while ϵz-greedy uses different random actions each time. As there are fewer obstacles in these two environments compared to Wall-maze, ϵz-greedy performs better.

The tree budget N upper bounds the maximum option length of ϵt-greedy due to the fact that the longest path between nodes in the density tree is shorter or equal to the number of nodes in the tree. The tree budget N has the same meaning as N in ϵz-greedy using uniform $z(n) = \mathbb{1}_{n \leq N}/N$ distribution.



(a) Wall-maze

(b) U-maze



(c) Point-push

Figure 5.4: The environment coverage for exploration strategies in three environments: (a) Wall-maze. (b) U-maze. (c) Point-push

Therefore, we assess the coverage for both methods with different budgets. We calculate the coverage for all budgets after 1 million training frames. Table 5.6 shows the results: ϵt-greedy using a perfect model achieves more coverage than ϵz-greedy with all budgets in all three environments. However, ϵt-greedy using the replay buffer has more coverage than ϵz-greedy only in Wall-maze, while it has similar coverage in the other two environments.

5.4 Effectiveness of Each New Component in ETGL-DDPG

We evaluated the performance of ETGL-DDPG as a whole. Now, we assess the impact of each component on DDPG separately. Figure 5.5 presents the results

budget N	ϵz-greedy			ϵt-greedy (perfect model)			ϵt-greedy (replay buffer)		
	Wall-maze	U-maze	Point-push	Wall-maze	U-maze	Point-push	Wall-maze	U-maze	Point-push
5	0.36	0.55	0.36	0.76	0.94	0.40	0.56	0.51	0.35
10	**0.38**	0.91	0.38	0.97	0.91	0.41	0.74	0.65	0.33
15	0.34	0.85	0.39	0.65	0.94	0.42	**1**	0.79	0.34
20	0.30	0.84	0.40	0.83	0.94	0.48	1	0.83	0.34
25	0.28	**0.86**	0.40	**1**	0.95	0.47	1	0.76	0.35
30	0.27	0.83	0.39	1	**0.97**	0.51	1	**0.85**	0.34
35	0.25	0.82	0.40	1	0.95	0.53	1	0.82	**0.37**
40	0.24	0.82	0.40	1	0.97	0.55	1	0.70	0.35
45	0.22	0.85	**0.41**	1	0.96	0.64	1	0.75	0.36
50	0.22	0.79	0.40	1	0.97	**0.73**	1	0.69	0.37

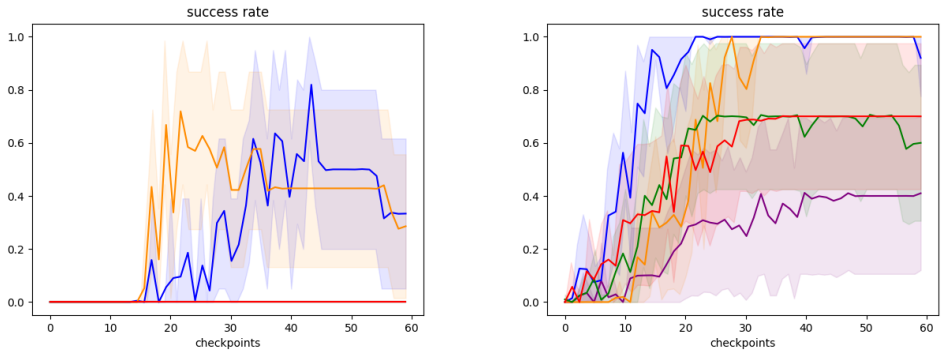
Table 5.6: Analysis of the impact of budget N on environment coverage.

for all environments. ϵt-greedy shows the most improvement, particularly with access to the perfect model. DDPG using ϵt-greedy achieves a success rate of 1 in Point-push and U-maze. In Wall-maze it cannot exceed a success rate of 0.6. In Wall-maze, both GDRB and longest n-step return failed to enhance the performance in the absence of ϵt-greedy, indicating the crucial role of our exploration strategy. In U-maze and Point-push, GDRB and longest n-step return improved the performance, although GDRB's impact is smaller in Point-push. We substitute reservoir sampling with FIFO as the retention policy in GDRB and obtain similar results.

Meng, Gorbet, and Kulić [24] used n from 1 to 8, the minimum of n-step updates, and their average. MMDDPG, which is the average of 1 to 8 steps (avg8-step), outperforms all other versions in robotic tasks. Here, we replace the longest n-step return with the average 8-step in ETGL-DDPG (perfect model) to see which one is more effective in reward propagation. Figure 5.6 shows the results for all three environments. MMDDPG and longest n-step return have similar performance but avg8-step converges faster than the longest n-step. This could be due to more stability resulting from taking the average of 8 updates.

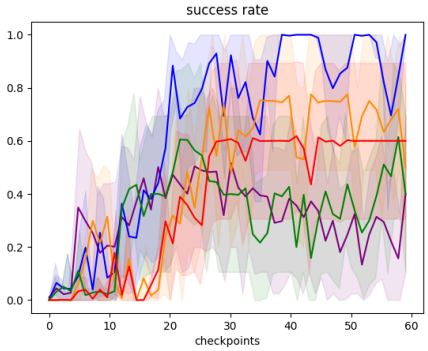
5.5 The Computational Perspective

ETGL-DDPG uses density trees to explore the environment. We empirically showed that this method has a better performance than simple techniques,



(a) Wall-maze

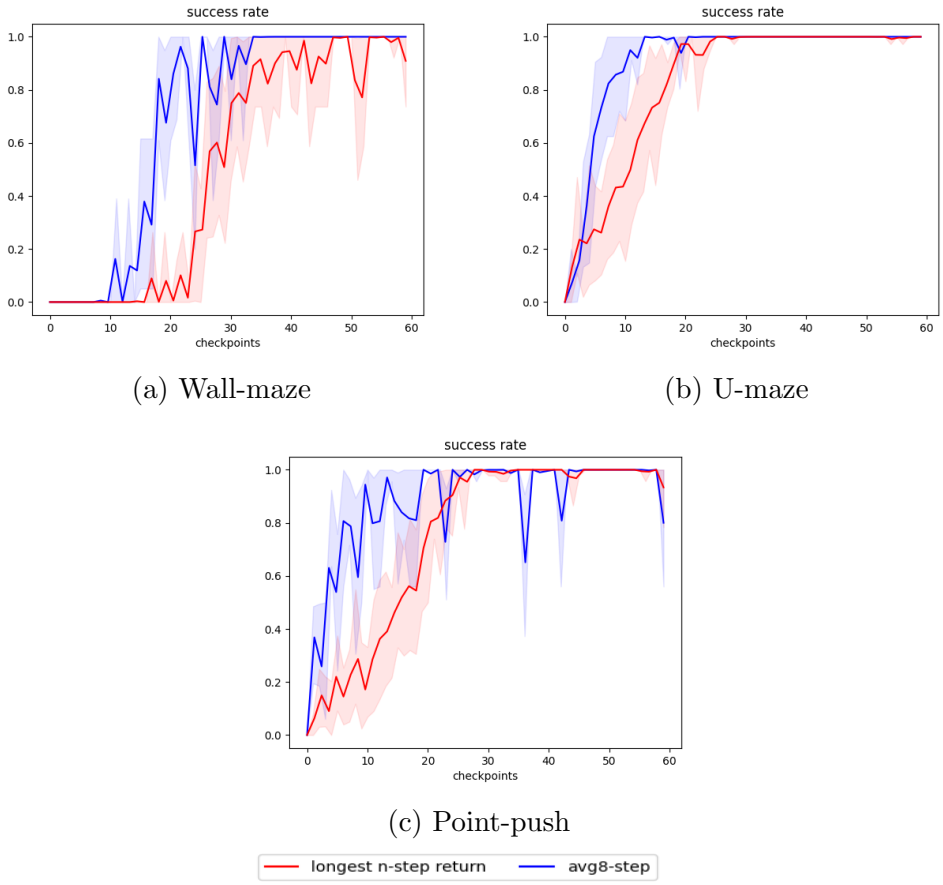
(b) U-maze



(c) Point-push

Figure 5.5: Analyzing the separate impact of four components on DDPG: ϵt-greedy (perfect model), ϵt-greedy (replay buffer), GRS-DRB, and longest n-step return. In Wall-maze, only ϵt-greedy versions could achieve a non-zero success rate.

such as injecting noise into the action space. However, during our experiments, we noticed that this method requires more computation, which leads to longer training times. We compared the training time of ETGL-DDPG with other baselines and found that it takes 1.5 times the clock time of DDPG. This implies that when using ETGL-DDPG for executing a training episode, it takes 1.5 times longer than the time taken for training with DDPG. Other baselines, including DDPG + HER, DDPG+ intrinsic motivation, and DDPG + ϵz-greedy, take almost the same clock time as DDPG. Moreover, if we want to build density trees in real-time, it takes twice as long as DDPG because we need to traverse the tree to add new nodes to it. If we assign the same



(a) Wall-maze

(b) U-maze

(c) Point-push

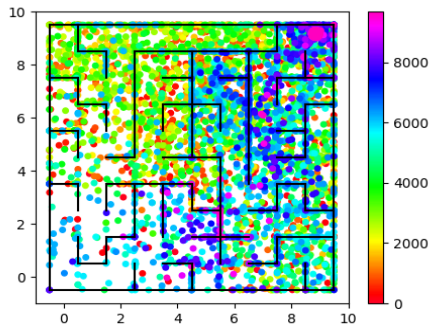
Figure 5.6: Comparing two multi-step TD update methods: the longest n-step return, and avg8-step (average of 1 to 8 steps)

computation of ETGL-DDPG to DDPG, it cannot improve its performance as the policy is stabilized and cannot visit new areas of state space.

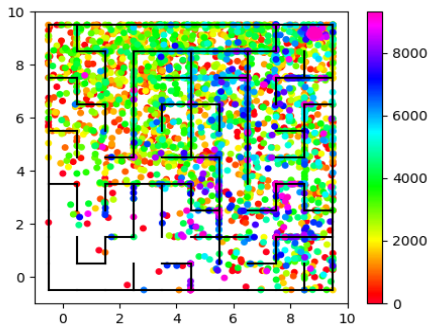
5.6 Distribution of Terminal States

In this section, we analyze the order in which the agent visits different parts of the environment by examining the distribution of the last states reached in each episode. To make it more visually appealing and easy to interpret, we only sample some of the episodes. The results for Wall-maze, U-maze, and Point-push can be found in Figures 5.7, 5.8, and 5.9, respectively. In Wall-maze, only ϵt-greedy versions can effectively navigate to different regions of the environment and ultimately reach the goal area. Other methods often get trapped in one of the local optima and are unable to reach the goal. The

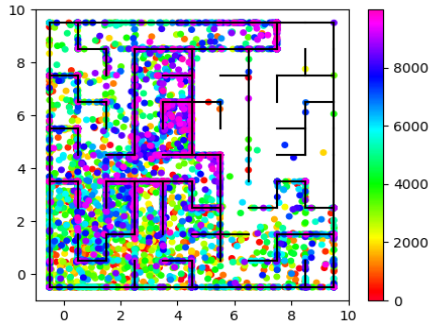
reason that some methods, such as DDPG + HER, seem to have fewer points is that the agent spends a lot of time revisiting congested areas instead of exploring new ones. In U-maze, most methods can explore the majority of the environment. However, during the final stages of training, methods such as DDPG, SAC, and DDPG + intrinsic motivation have lower success rates and may end up focusing on locations away from the goal areas. In Point-push, ϵt-greedy versions, ϵz-greedy, and DDPG + HER first visit the lower section of the environment in the early stages. After that, they push aside the movable box and proceed to the upper section to visit the goal area. For the other methods, the pattern is almost the same, with occasional visits to the lower section.



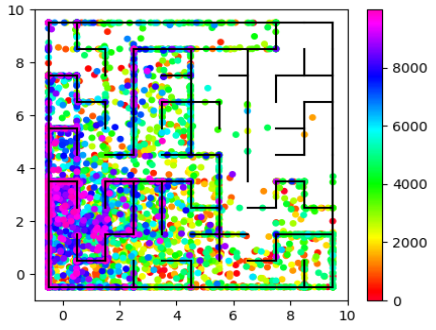
(a) DDPG + ϵt-greedy (perfect model)



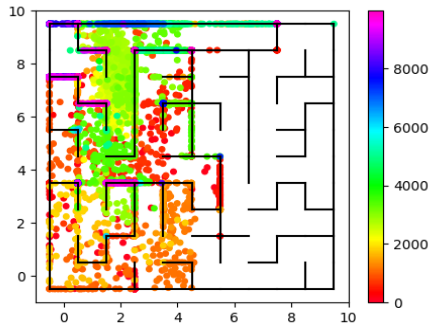
(b) DDPG + ϵt-greedy (replay buffer)



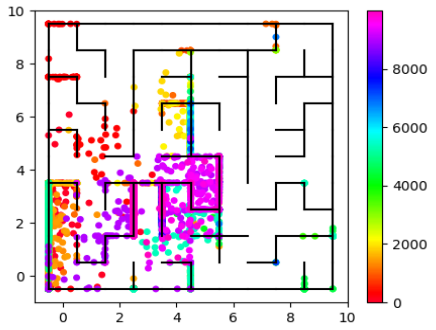
(c) DDPG + ϵz-greedy



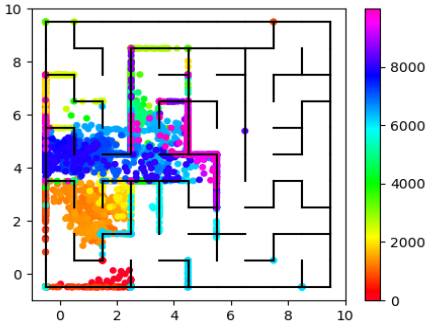
(d) SAC



(e) DDPG + intrinsic motivation

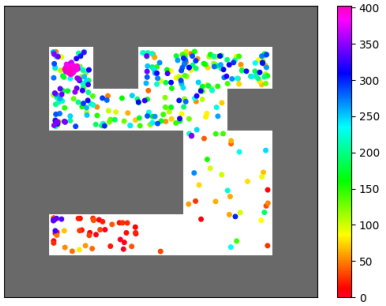


(f) DDPG + HER

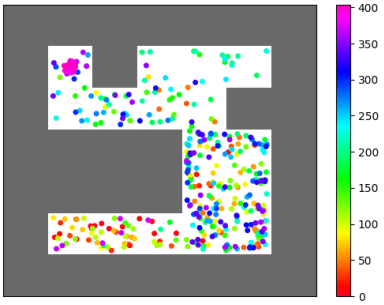


(g) DDPG

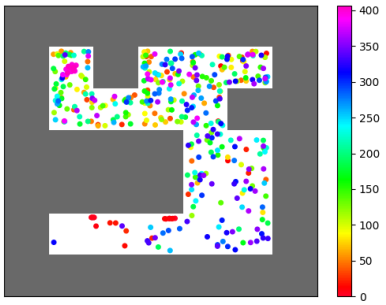
Figure 5.7: The agent's location at the end of episodes throughout the training in Wall-maze.



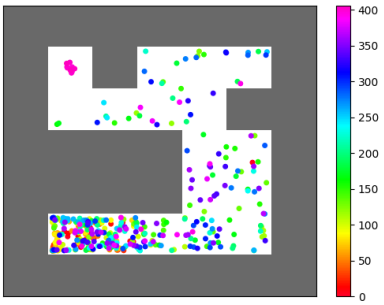
(a) DDPG + ϵt-greedy (perfect model)



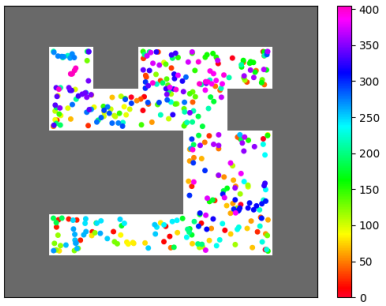
(b) DDPG + ϵt-greedy (replay buffer)



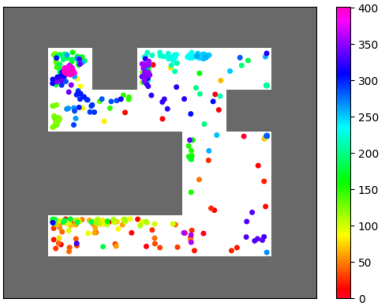
(c) DDPG + ϵz-greedy



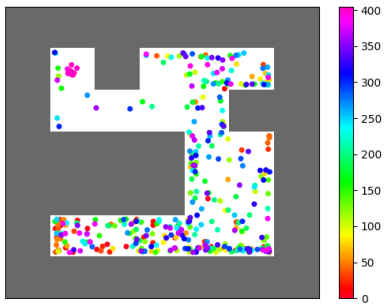
(d) SAC



(e) DDPG + intrinsic motivation

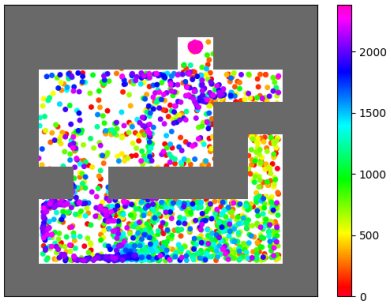


(f) DDPG + HER

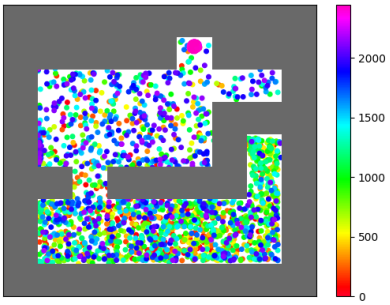


(g) DDPG

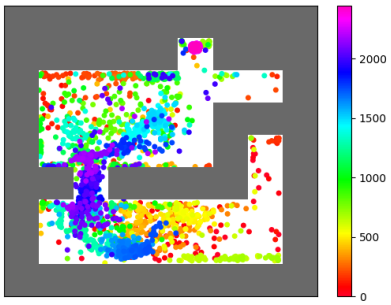
Figure 5.8: The agent's location at the end of episodes throughout the training in U-maze.



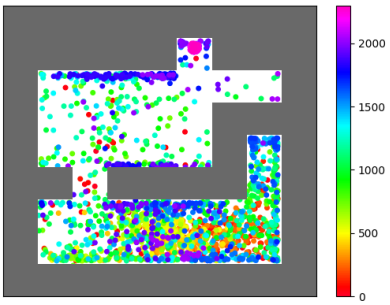
(a) DDPG + ϵt-greedy (perfect model)



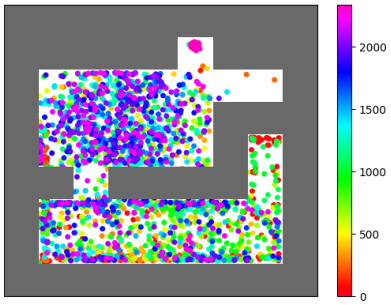
(b) DDPG + ϵt-greedy (replay buffer)



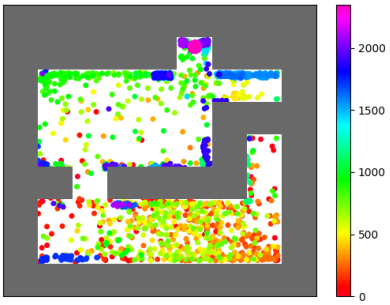
(c) DDPG + ϵz-greedy



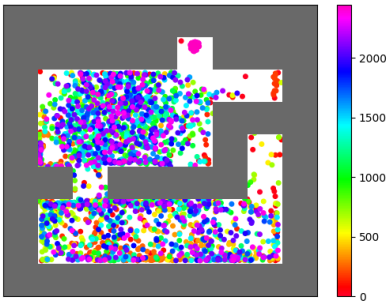
(d) SAC



(e) DDPG + intrinsic motivation



(f) DDPG + HER



(g) DDPG

Figure 5.9: The agent's location at the end of episodes throughout the training in Point-push.

Chapter 6

Conclusion

We have introduced three components that improve the performance of the DDPG algorithm in sparse-reward goal-conditioned environments. ϵt-greedy is a temporally extended version of ϵ-greedy using options generated by search. We prove that ϵt-greedy achieves a polynomial sample complexity under specific MDP structural assumptions. While the basic form of ϵt-greedy requires a perfect model of the environment, which limits its applicability, we show that we can use the replay buffer to achieve comparable results. GDRB employs an extra buffer to separately store successful episodes. The longest n-step return bootstraps from the Q-value of the final state in unsuccessful episodes and becomes a Monte Carlo update for successful episodes.

ETGL-DDPG uses these components with DDPG and outperforms state-of-the-art methods, at the expense of about 1.5x wall-clock time w.r.t DDPG. One limitation for this work is that we approximate visit counts by discretizing the environment and use the L^2 norm to group observed transitions. For high-dimensional state spaces, one future direction is to apply *normalizing flows* [29] and techniques such as *simhash* [7] for density estimation. Also, it would be interesting to analyze the use of ϵt-greedy in scenarios other than goal-conditioned tasks such as Atari games [26], since it does not depend on the position of the goal, similar to ϵz-greedy.

References

[1] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi, "Solving the rubik's cube with deep reinforcement learning and search," *Nature Machine Intelligence*, vol. 1, no. 8, pp. 356–363, 2019.

[2] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, OpenAI, and W. Zaremba, "Hindsight experience replay," *Advances in neural information processing systems*, vol. 30, 2017.

[3] A. P. Badia, P. Sprechmann, A. Vitvitskyi, D. Guo, B. Piot, S. Kapturowski, O. Tieleman, M. Arjovsky, A. Pritzel, A. Bolt, *et al.*, "Never give up: Learning directed exploration strategies," in *International Conference on Learning Representations*, 2019.

[4] M. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," *Advances in neural information processing systems*, vol. 29, 2016.

[5] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[6] Y. Burda, H. Edwards, A. Storkey, and O. Klimov, "Exploration by random network distillation," in *International Conference on Learning Representations*, 2018.

[7] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, 2002, pp. 380–388.

[8] W. Chen and M. Müller, "Continuous arvand: Motion planning with monte carlo random walks," in *ICAPS 2015 Workshop on Planning and Robotics (PlanRob)*, 2015, pp. 23–29.

[9] C. Colas, O. Sigaud, and P.-Y. Oudeyer, "GEP-PG: Decoupling exploration and exploitation in deep reinforcement learning algorithms," in *International Conference on Machine Learning*, PMLR, 2018, pp. 1039–1048.

[10] W. Dabney, G. Ostrovski, and A. Barreto, "Temporally-extended ε-greedy exploration," in *International Conference on Learning Representations*, 2020.

[11] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, *OpenAI baselines*, `https://github.com/openai/baselines`, 2017.

[12] B. Eysenbach, R. R. Salakhutdinov, and S. Levine, "Search on the replay buffer: Bridging planning and reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[13] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International Conference on Machine Learning*, PMLR, 2018, pp. 1861–1870.

[14] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. DOI: `10.1109/tssc.1968.300136`. [Online]. Available: `https://doi.org/10.1109/tssc.1968.300136`.

[15] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, 2018.

[16] Y. Hu, W. Wang, H. Jia, Y. Wang, Y. Chen, J. Hao, F. Wu, and C. Fan, "Learning to utilize shaping rewards: A new approach of reward shaping," *Advances in Neural Information Processing Systems*, vol. 33, pp. 15 931–15 941, 2020.

[17] Y. Kanagawa, *Mujoco-maze*, `https://github.com/kngwyu/mujoco-maze`, 2021.

[18] A. D. Laud, *Theory and application of reward shaping in reinforcement learning*. University of Illinois at Urbana-Champaign, 2004.

[19] S. LaValle, "Rapidly-exploring random trees: A new tool for path planning," *Research Report 9811*, 1998.

[20] J. Lehman, J. Chen, J. Clune, and K. O. Stanley, "ES is more than just a traditional finite-difference approximator," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018, pp. 450–457.

[21] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *International Conference on Learning Representations*, 2016.

[22] Y. Liu and E. Brunskill, "When simple exploration is sample efficient: Identifying sufficient conditions for random exploration to yield PAC RL algorithms," *CoRR*, vol. abs/1805.09045, 2018.

[23] G. Matheron, N. Perrin, and O. Sigaud, "Understanding failures of deterministic actor-critic with continuous action spaces and sparse rewards," in *International Conference on Artificial Neural Networks*, Springer, 2020, pp. 308–320.

[24] L. Meng, R. Gorbet, and D. Kulić, "The effect of multi-step methods on overestimation in deep reinforcement learning," in *2020 25th International Conference on Pattern Recognition (ICPR)*, IEEE, 2021, pp. 347–353.

[25] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, PMLR, 2016, pp. 1928–1937.

[26] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[28] G. Ostrovski, M. G. Bellemare, A. Oord, and R. Munos, "Count-based exploration with neural density models," in *International Conference on Machine Learning*, PMLR, 2017, pp. 2721–2730.

[29] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, and B. Lakshminarayanan, "Normalizing flows for probabilistic modeling and inference," *The Journal of Machine Learning Research*, vol. 22, no. 1, pp. 2617–2680, 2021.

[30] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," in *International Conference on Machine Learning*, PMLR, 2017, pp. 2778–2787.

[31] L. Schäfer, F. Christianos, J. P. Hanna, and S. V. Albrecht, "Decoupled reinforcement learning to stabilise intrinsically-motivated exploration," *21st International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2022, Auckland, New Zealand, May 9-13, 2022*, 2021.

[32] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *International Conference on Learning Representations*, 2016.

[33] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[34] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *International Conference on Machine Learning*, PMLR, 2014, pp. 387–395.

[35] S. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvári, "Convergence results for single-step on-policy reinforcement-learning algorithms," *Machine learning*, vol. 38, pp. 287–308, 2000.

[36] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[37] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[38] S. Thrun and A. Schwartz, "Issues in using function approximation for reinforcement learning," in *Proceedings of the Fourth Connectionist Models Summer School*, Hillsdale, NJ, vol. 255, 1993, p. 263.

[39] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2012, pp. 5026–5033.

[40] A. Trott, S. Zheng, C. Xiong, and R. Socher, "Keeping your distance: Solving sparse reward tasks using self-balancing shaped rewards," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[41] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the Brownian motion," *Physical review*, vol. 36, no. 5, p. 823, 1930.

[42] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[43] L. Zhang, Z. Zhang, Z. Pan, Y. Chen, J. Zhu, Z. Wang, M. Wang, and C. Fan, "A framework of dual replay buffer: Balancing forgetting and generalization in reinforcement learning," in *Proceedings of the 2nd Workshop on Scaling Up Reinforcement Learning (SURL), International Joint Conference on Artificial Intelligence (IJCAI)*, 2019.

Appendix

In this appendix, we show the proof of Theorem 1, which is mostly due to Shayan Karimi.

Theorem 1. *Given S as state space and A as action space, and a set of options Ω_T generated by density trees after some number of explorations. Suppose tree budget $N \leq \Theta(|S||A|)$, and sampling distribution $\Pr[\omega]$, which $\omega \in \Omega_T$, is defined over generated options. If $\Pr[\omega] \geq \frac{1}{\Theta(|S||A|)}$, ϵt-greedy can achieve a polynomial sample complexity.*

Proof. Based on the paper by [22], and the analysis of the covering length when following a random policy, we have the following proposition:

Preposition 6.0.1. *(Corollary 14 of the paper [22]) : For any irreducable MDP M, we define $P_{\pi_{RW}}$ as a transition matrix induced by random walk policy π_{RW} over M and $L(P_{\pi_{RW}})$ is denoted as the Laplacian of this transition matrix. Suppose λ is the smallest non-zero eigenvalue of L and $\Psi(s)$ is the stationary distribution over states which is induced by random walk policy, then Q-learning with random walk exploration is a PAC RL algorithm if: $\frac{1}{\lambda}, \frac{1}{\min_s \Psi(s)}$ are Poly($|S||A|$).*

Note that Proposition 6.0.1 is not limited to an MDP with primitive actions. Therefore, we can broaden its scope by incorporating options into this proposition and demonstrate that both $\frac{1}{\lambda}$ and $\frac{1}{\min_s \Psi(s)}$ can be polynomially bounded in terms of MDP parameters—in this case, states and actions in our approach.

Let's begin by examining the upper-bound for $\frac{1}{\min_s \Psi(s)}$. Suppose we are at the ith exploration phase within the t_i exploration tree. In this tree, let's

designate S_{root} as the state assigned as the root of the tree during the exploration phase. Now, consider another random state (excluding S_{root}) within this tree structure, denoted as S_{rand}. We acknowledge that, when considering the entire state space, there can be multiple options constructed from S_{root} to S_{rand}. Each tree t_i provides one of these options, and due to the fact that $\Psi(s)$ is defined over all states and ω is the option with a limited size because of the constrained tree budget, we can calculate the upper-bound for $\frac{1}{\min_s \Psi(s)}$ as follows:

$$\begin{aligned} \Psi(S_{rand}) = \sum_{\omega \in \Omega_T} \Pr[\omega]\Psi(S_{root}) \Rightarrow \Psi(S_{rand}) \geq \Pr[\omega]\Psi(S_{root}), \\ \frac{1}{\Psi(S_{rand})} \leq \frac{1}{\Pr[\omega]} \frac{1}{\Psi(S_{root})} \Rightarrow \frac{1}{\Psi(S_{rand})} \leq \frac{\Theta(|S||A|)}{\Psi(S_{root})} \end{aligned} \tag{6.1}$$

Since S_{rand} can represent any of the states encountered in a tree, we can state:

$$\frac{1}{\Psi(S_{rand})} \leq \frac{\Theta(|S||A|)}{\Psi(S_{root})} \Rightarrow \frac{1}{\min_s \Psi(S_{rand})} \leq \frac{\Theta(|S||A|)}{\Psi(S_{root})} \tag{6.2}$$

So, $\frac{1}{\min_s \Psi(s)}$ is polynomially bounded. Now, we need to demonstrate that $\frac{1}{\lambda}$ is also polynomially bounded. To bound λ, we first need to recall the definition of the Cheeger constant, h. Drawing from graph theory, if we denote $V(G)$ and $E(G)$ as the set of vertices and edges of an undirected graph G, respectively, and considering the subset of vertices denoted by V_s, we can define σV_s as follows:

$$\sigma V_s := \{(n_1, n_2) \in E(G) : n_1 \in V_s, n_2 \in V(G) \setminus V_s\} \tag{6.3}$$

So, σV_s can be regarded as a collection of all edges going from V_s to the vertex set outside of V_s. In the above definition, (n_1, n_2) is considered as a graph edge. Now, we can define a Cheeger constant:

$$h(G) := \min\{\frac{|\sigma V_s|}{|V_s|} : V_s \subseteq V(G), 0 < V_s \leq \frac{1}{2}|V(G)|\} \tag{6.4}$$

We are aware that $h \geq \lambda \geq \frac{h^2}{2}$, and by polynomially bounding h, we can ensure that λ is also bounded. In a related work [22], an alternative variation

of the Cheeger constant is utilized, which is based on the flow F induced by the stationary distribution Ψ of a random walk on the graph. Suppose for nodes n_1, n_2 and subset of nodes N_1 in the graph, we have:

$$F(n_1, n_2) = \Psi(n_1)P(n_1, n_2), \tag{6.5}$$

$$F(\sigma N_1) = \sum_{n_1 \in N_1, n_2 \notin N_1} F(n_1, n_2), \tag{6.6}$$

$$F(N_1) = \sum_{n_1 \in N_1} \Psi(n_1) \tag{6.7}$$

Building upon the aforementioned definition, the Cheeger constant is defined as:

$$h := \inf_{N_1} \frac{F(\sigma N_1)}{\min\{F(N_1), F(\bar{N_1})\}} \tag{6.8}$$

Suppose $N_{rand} = \{S_{root}\}$; we will now demonstrate that $\frac{1}{h}$ can be polynomially bounded:

$$h = \inf_{N_1} \frac{F(\sigma N_1)}{\min\{F(N_1), F(\bar{N_1})\}} \geq \frac{F(\sigma N_{rand})}{\min\{F(N_{rand}), F(\overline{N_{rand}})\}} \geq \frac{\sum_{s \neq S_{root}} \Psi(S_{root})P(S_{root}, s)}{\Psi(S_{root})},$$
$$= \sum_{s \neq S_{root}} P(S_{root}, s) \geq \Pr[\omega] \Rightarrow \frac{1}{h} \leq \Theta(|S||A|)$$

We demonstrate that both terms stated in Proposition 6.0.1 are polynomially bounded, and thus, the proof is complete. It is worth noting that the proof sketch is similar to the one given for ϵz-greedy [10], as it also has a polynomial sample complexity given the same conditions. □