Exploiting Reward Machines with Deep Reinforcement Learning in Continuous Action Domains

Haolin Sun

A DISSERTATION SUBMITTED TO
THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF COMPUTER SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

January 2023

©Haolin Sun, 2023

Abstract

Deep reinforcement learning can solve real-world robot control problems, such as autonomous driving and robotic arm manipulation. The goal of a deep reinforcement learning agent is to find an optimal strategy for each specific task. In deep reinforcement learning, an agent does not know the problem description and learns the optimal solution through trial-and-error. This method brings two major challenges when solving real-world problems: partial observability and learning efficiency. In this thesis, we address these two challenges and extend previous work. First, we use reward machines to address the problem of partial observability. A reward machine can decompose a task and pass its high-level structure to the agent, letting the agent exploit this information. It supports non-Markovian task specifications, thus accommodating some partial observability. Then, we focus on finding the existing cutting-edge deep reinforcement learning algorithms and integrating them with reward machines to enhance the learning efficiency. To test the performance of all the algorithms, we proposed a series of different tasks that can be used to mimic real-world robot control problems. Finally, based on the test results, we compare the performance of all the algorithms and analyze their advantages and disadvantages.

To Yongmei Cui, Wentao Sun and Chang Liu

Acknowledgments

First, I would like to express my most sincere gratitude to my supervisor, Professor Yves Lespérance. Thank you for providing me with the opportunity to do research with you. To be honest, I did not expect that I would get an offer when I applied to graduate studies because my undergraduate GPA was not that outstanding. However, you chose to believe in me as a seemingly unimpressive student, and you made my dream come true. Your decision completely changed my life, giving me access to the field of artificial intelligence, which is one of the most trending topics right now. You truly helped me expand my knowledge; you even made me feel more confident in my life. During the time when I was looking for a topic for my thesis, I felt panicked and confused, but you were always there to provide me with various research directions and inspiration, which helped me get through that most difficult time. I appreciate you from the bottom of my heart for your guidance and support, and I look forward to continuing to explore the field of AI with you in the future.

I would also like to express my appreciation to my internal committee member, Professor Frank van Breugel. Thank you very much for your valuable comments and suggestions on my thesis. The questions you asked were crucial, and they helped me to think more deeply about many details of my thesis and made my understanding of it more thorough.

Finally, a special thanks to my external member, Professor Jinjun Shan, for taking the time to attend my defence while attending the conference. Thank you very much for your time and support.

Contents

Abstract **ii**

Dedication **iii**

Acknowledgments **iv**

Contents **v**

List of Figures **viii**

1 Introduction **1**

1.1 Motivation . . . 3

1.2 Our Approach and Contribution . . . 3

1.2.1 The Combination of Reward Machines and Popular Deep RL Algorithms . . . 3

1.2.2 Defining Various Types of Tasks and Validating Algorithm Performance . . . 4

1.2.3 Contributions . . . 4

1.3 Outline of the Thesis . . . 5

2 Background **6**

2.1 Discrete and Continuous Action Domains . . . 6

2.2 Deterministic vs Stochastic Policies and their Representation . . . 7

2.2.1 Stochastic Policy Output in Discrete Action Domains . . . 8

2.2.2 Stochastic Policy Output in Continuous Action Domains . . . 8

2.3 Exploration and Exploitation . . . 8

2.4 Decaying ϵ-greedy Method . . . 9

2.5 Sampling and KL Divergence . . . 9

2.6 On-policy and Off-policy Update . . . 10

2.7 Actor-Critic Architecture . . . 11

3 Related Work 12

3.1 Reward Machines, CRM and HRM . . . 12

3.1.1 Reward Machines . . . 12

3.1.2 Counterfactual Experience for reward machines (CRM) . . . 14

3.1.3 Hierarchical Reinforcement Learning for Reward Machines (HRM) . . . 17

3.2 Deep Deterministic Policy Gradient (DDPG) . . . 18

3.3 DDPG with CRM . . . 20

3.4 Maximum Entropy Reinforcement Learning and Soft Actor-Critic (SAC) . . . 22

3.4.1 Maximum Entropy Reinforcement Learning (MERL) . . . 22

3.4.2 Soft Actor-Critic (SAC) . . . 23

3.5 Twin Delayed Deep Deterministic Policy Gradient (TD3) . . . 29

3.5.1 Several Improvements of TD3 based on DDPG . . . 29

3.5.2 TD3 implementation . . . 31

3.6 Proximal Policy Optimization (PPO) . . . 31

3.6.1 Stochastic Gradient Ascent . . . 33

3.6.2 Trust Region . . . 35

3.6.3 PPO Implementation . . . 36

4 New Algorithms for Reward Machines in Deep Reinforcement Learning 39

4.1 Soft Actor-Critic (SAC) with CRM . . . 39

4.2 Twin Delayed Deep Deterministic Policy Gradient (TD3) with CRM . . . 42

4.3 Proximal Policy Optimization (PPO) with CRM . . . 44

4.4 Discussion . . . 46

5 Experimental Evaluation 48

5.1 Results in the Half-Cheetah Domain . . . 49

5.1.1 Performance on Task 1 and Task 2 . . . 52

5.1.2 Performance on Task 3 . . . 57

5.1.3 Performance on Task 4 . . . 58

5.1.4 Performance on Task 5 . . . 59

5.2 Results in the Ant Domain . . . 61

5.2.1 Performance on Task 1 and Task 2 . . . 63

5.2.2 Performance on Task 3 65
5.3 Discussion 66
5.3.1 Overall Performance of SAC-CRM 66
5.3.2 Overall Performance of DDPG-CRM 67
5.3.3 Overall Performance of Option-DHRM 67
5.3.4 Overall Performance of TD3-CRM 68
5.3.5 Overall Performance of PPO-CRM 68

6 Conclusion **70**
6.1 Summary of Contributions 70
6.2 Future Work 71
6.3 Concluding Remarks 73

Bibliography **74**

List of Figures

3.1 An example RM environment 13

3.2 The automaton of the RM environment example 13

3.3 A multimodal Q-function 25

5.1 The Half-Cheetah domain 49

5.2 The RM automaton of Task 1 in the Half-Cheetah domain. 50

5.3 The RM automaton of Task 2 in the Half-Cheetah domain. 50

5.4 The RM automaton of Task 3 in the Half-Cheetah domain. 51

5.5 The RM automaton of Task 4 in the Half-Cheetah domain. 51

5.6 The RM automaton of Task 5 in the Half-Cheetah domain. 51

5.7 CRM vs baselines in Half-Cheetah Task 1 53

5.8 CRM vs baselines in Half-Cheetah Task 2 54

5.9 Results for Task 1 in the Half-Cheetah domain 54

5.10 Results for Task 2 in the Half-Cheetah domain 55

5.11 PPO-CRM vs baseline PPO in Half-Cheetah Task 1 56

5.12 PPO-CRM vs baseline PPO in Half-Cheetah Task 2 57

5.13 Results for Task 3 in the Half-Cheetah domain 58

5.14 Results for Task 4 in the Half-Cheetah domain 59

5.15 Results for Task 5 in the Half-Cheetah domain 60

5.16 The Ant domain 62

5.17 The RM automaton of Task 3 in the Ant domain. 63

5.18 Results for Task 1 in the Ant domain 63

5.19 Results for Task 2 in the Ant domain 64

5.20 Results for Task 3 in the Ant domain 65

Chapter 1

Introduction

In artificial intelligence, an intelligent agent must make appropriate decisions and actions at each moment to complete a given task. For example, in autonomous driving, the vehicle needs to decide the current driving behavior based on the current road conditions to ensure safe arrival at the destination; for a robotic arm to grasp objects, the arm needs to determine how to drive itself to grasp the object quickly and accurately. The critical point of solving such problems is how to make the best decisions and execute the best actions to achieve the desired goal in a constantly changing environment. The learning approach to solving such problems is called *reinforcement learning (RL)*.

In reinforcement learning, an agent interacts with the environment by performing an action in each environmental state, with each interaction resulting in a reward signal. The goal of an RL agent is to learn a policy (a mapping from observations to actions) that maximizes the expected cumulative reward and to improve its policy from past learning experiences.

In any environment state, an RL agent does not pursue an action that can bring the maximum reward only in this state, but instead pursues an action that can bring a larger expected reward throughout the task. When choosing an action, the agent usually takes a greedy approach, i.e., prefers the action that has been tried in the history of interaction and can bring more expected benefits. However, untried actions may have higher expected rewards. Therefore, the agent needs to balance the frequency of exploiting the known actions and exploring new actions to find the optimal policy quickly and accurately.

In some simple discrete action domains, such as simple turn-based games, where the number of states and actions is finite, a basic RL algorithm such as Q-learning [28] will be sufficient to find the optimal policy relatively quickly. The core idea of Q-learning is straightforward, i.e., try all the possible actions in each state and record their expected rewards. When the same state is encountered in the next round of learning,

the agent will select the action with the highest expected reward to obtain the optimal policy. However, in a more complex continuous action domain, such as autonomous driving, because the values of acceleration, deceleration, and steering wheel rotation are from infinite domains, the agent cannot try all possible actions during the learning process, so Q-learning will not be able to identify the actions with the highest expected rewards in each state and thus cannot find the optimal policy.

Deep reinforcement learning was developed in part to solve the RL problems in continuous action domains. Deep reinforcement learning integrates the powerful understanding capability of neural networks and the decision-making capability of reinforcement learning, enabling agents to solve more complex problems in continuous action domains [9].

The reward functions in the RL algorithms are usually a black box, where an RL agent receives the reward only when it completes the whole task, and the agent does not know what conditions are necessary to be triggered in the process of completing the task. As such, if the agent wants to learn the optimal policy, it must interact with the environment on an ongoing basis, and the interaction process takes a lot of time and computational power. However, if the agent can access the internal structure of the reward function and thus understand the high-level idea of the task, then the agent can exploit this information to speed up the learning of the optimal policy.

In order to let the agent have access to the reward function, Toro Icarte et al. proposed a finite state machine, called the *reward machine (RM)* [25], which defines a novel form for the reward function. Reward machines can define multiple forms of reward functions, including concatenation, loops, and conditional rules. A reward machine can decompose a complex task into multiple subtasks and expose the corresponding reward function in each subtask to the agent, thus allowing the agent to learn the reward function at each RM state. For each RM state transferred, the reward of the specific subtask will be returned, enabling agents to learn RM state by RM state, thus allowing the agent to conduct less exploration and speed up the learning. Reward machines are flexible in expression; tasks can be expressed in a variety of ways, such as Linear Temporal Logic (LTL) [8] or other formal languages, and then translated into a reward machine.

Meanwhile, in order to exploit the structure of a reward machine, Toro Icarte et al. proposed a novel approach called counterfactual experiences for reward machines (CRM) [23]. When the agent interacts with the environment, CRM uses the information from the reward functions provided by reward machines to generate synthetic experiences to help the agent make more explicit judgments about RM states, thus accelerating the agent's learning speed.

1.1 Motivation

Reward machines can be used in both discrete and continuous action domains. In discrete action domains, Toro Icarte et al. have improved the learning efficiency of the existing RL algorithms by combining CRM with Q-learning and Double DQN [26], where RM-based Q-learning can converge to the optimal policy. However, the only existing algorithms that have been combined with reward machines in continuous action domains are DDPG [15] and option-based Hierarchical Reinforcement Learning (HRL) [20]. With the emergence of new deep RL algorithms, the advantages of DDPG and option-based HRL have become less prominent. In some environments, the performance of other newly proposed deep RL algorithms has surpassed that of DDPG and option-based HRL. In order to further improve the learning efficiency of RM-based algorithms in various continuous action domains, we will work on two things in this thesis:

1. **Adapting other deep RL algorithms to work with RMs.** That is, trying to combine reward machines with other newer deep RL algorithms and aiming to make them perform better than DDPG and option-based HRL in various continuous domains.

2. **Defining various types of benchmark tasks and using them to evaluate our algorithms.** That is, defining various types of tasks in continuous action domains to explore the performance and use cases of different RM-based algorithms.

1.2 Our Approach and Contribution

In this thesis, we combine CRM with several popular deep RL algorithms, including Soft Actor-Critic (SAC) [10], Twin-Delayed Deep Deterministic Policy Gradient (TD3) [7], and Proximal Policy Optimization (PPO) [18]. These new algorithms, as well as the DDPG-CRM and HRM proposed by Toro Icarte et al., are tested and compared in eight tasks in two continuous domains. We also summarize the strengths and weaknesses of each algorithm with their reasons.

1.2.1 The Combination of Reward Machines and Popular Deep RL Algorithms

The role of reward machines is to decompose a complex task into multiple subtasks and expose the reward function of each subtask to the algorithms, allowing the algorithms to obtain more explicit information from it. As such, the learning ability of the algorithm itself will directly affect whether the agent can get a high reward value.

In order to further improve the learning efficiency of RM-based algorithms in continuous action domains, we combined reward machines with popular deep RL algorithms proposed in recent years, including Soft

Actor-Critic (SAC), Twin-Delayed Deep Deterministic Policy Gradient (TD3), and Proximal Policy Gradient (PPO). These algorithms outperformed DDPG in many previous experimental studies, so we expect that they will outperform DDPG-CRM in the same environments when combined with CRM.

1.2.2 Defining Various Types of Tasks and Validating Algorithm Performance

Our experimental section validates the performance of all the algorithms by defining a total of eight tasks in two different continuous action domains. Among them, five tasks involve a 2D environment, and three involve a 3D environment. For the 2D environment, we choose one of the mainstream test environments, *Half-Cheetah* [3], which was developed by Open AI. Our Half-Cheetah environment contains five tasks, including two tasks defined by Toro Icarte et al. [23] and three new tasks defined by ourselves. These tasks are of different types and complexity, allowing a more comprehensive evaluation of the performance of all algorithms in this environment. In addition, to test the performance of the algorithms in a more complex environment, we add *Ant* [3] as another environment, which was also developed by Open AI. Ant is a 3D environment with a higher complexity than the Half-Cheetah, and we define a total of three tasks in this environment.

1.2.3 Contributions

The main contributions of this thesis can be summarized as follows.

First, we combined CRM with several mainstream deep RL algorithms for which the combination had not been studied before. By researching the current mainstream deep RL algorithms, we selected three widely used and well-performing deep RL algorithms to combine with CRM. The combined algorithms are called SAC-CRM, TD3-CRM, and PPO-CRM.

Then, we added more task types compared to previous experiments, e.g., [25]. Based on the RM model, we defined a total of six new tasks in different continuous action domains to further validate the performance of all RM-based algorithms in different environments and task types. Based on the experimental results, we compared the performance of all the RM-based deep RL algorithms in different environments and tasks and analyzed the reasons for the performance difference.

Last but not least, we discovered a new, better-performing CRM-based deep RL algorithm. Through a series of experimental validations, we found that in SAC-CRM, the learning speed and reward values it achieves within the specified number of learning steps are the best among all the algorithms for most tasks.

1.3 Outline of the Thesis

The rest of the thesis is organized as follows: Chapter 2 explains some essential concepts in reinforcement learning and discusses some key terms that appear in some of the algorithms. Chapter 3 comprehensively explains all the existing algorithms that we build on in this thesis, listing and explaining the unique learning mechanisms and their roles in each algorithm. Chapters 4 and 5 present the main contributions of this thesis. Chapter 4 explains the rationale and process of combining the new algorithms with reward machines and shows how to add counterfactual experiences to typical deep RL algorithms. Chapter 5 presents our experimental evaluation of the algorithms, containing descriptions of all eight tasks, which include two tasks defined by Toro Icarte et al. [25] and six brand-new tasks. Chapter 5 also shows the performance of all the algorithms and their corresponding analyses. Finally, Chapter 6 summarizes the contributions of this thesis and discusses future work.

Chapter 2

Background

In this chapter, we will explain key RL concepts and terms that appear in this thesis. Reinforcement learning has a wide range of use cases, so the RL environments are diverse. As such, the algorithms applied in different RL environments are different. Moreover, the ways in which each algorithm updates its policies are different, and these differences can affect the resulting policies. This chapter highlights the fundamental concepts of the RL algorithms used in the thesis to help the reader better understand the theory behind all the algorithms.

2.1 Discrete and Continuous Action Domains

The action domain is an essential setting in RL problems, and each agent needs to be trained in a specific type of action domain. The design of the action domain is crucial for the choice of RL algorithms. In different types of action domains, the agent needs to adopt different RL algorithms to maximize the reward; meanwhile, the complexity of the action domain also significantly impacts the performance of RL algorithms. There are two main classes of action domains in reinforcement learning: discrete action domains and continuous action domains [5].

In a discrete action domain, the possible choice of an action is drawn from a finite number of discrete values. For example, in the Snake game, the available actions set is $A = \{left, right, up, down\}$, all the actions will be selected from these four actions. In contrast, in a continuous action domain, the possible set of choices for an action is infinite, which is usually represented by a real vector, such as $A \in [-1, 1]$. For example, when a robot moves with human-like limbs, its knee joints have to make corresponding movements to drive the movement of the legs, including the rotation angle and the rotation force of the joints. The units such as angle and strength can be freely chosen within a continuous range and cannot be represented

by a finite number of discrete numerical values. An RL environment with this type of action domain is called a continuous action domain. Note that in a continuous action domain, the continuous actions can be discretized, and previous work has shown that using discretized actions with some on-policy algorithms (such as PPO) can make learning more efficient [22]. However, very little related work has been done so far on the discretization of continuous actions, and the existing studies cannot adequately demonstrate the broad applicability of discretized actions. Therefore, the default choice for the vast majority of deep RL algorithms is still to sample from the continuous action distributions.

2.2 Deterministic vs Stochastic Policies and their Representation

In reinforcement learning, the goal of an agent in a task is to learn a good policy to help it choose the best action among different states to obtain the highest possible reward value. Specifically, there are two forms of policy representation: deterministic and stochastic. They are represented differently in the actual neural network's output layer and thus influence how the agent selects actions. Intuitively, a deterministic policy is a decisive policy whose output for a given state is a fixed action to be performed by an agent. Formally, a deterministic policy π is a function from state to action. For an input state s, the output action of a deterministic policy can be represented as:

$$a = \pi(s) \tag{2.1}$$

In contrast, the output of a stochastic policy is not an unique action, but the probability that the agent will take an action for all possible actions at a given state. A stochastic policy can be represented as:

$$\pi(s) = P[a \mid s] \tag{2.2}$$

where, i.e., formally a stochastic policy π is a function from states to probability distributions over actions. As such, the input is still a state, but instead of the output being an action, it is the probability distribution of all possible actions.

Generally, an RL policy can be represented by a neural network, where the input is a specific environmental state. In a deterministic policy, the output layer will contain only one node, representing the optimal action derived from the policy. In contrast, the output layer of a stochastic policy will not be a single action.

2.2.1 Stochastic Policy Output in Discrete Action Domains

In discrete action domains, since the number of available actions is finite, the number of nodes in the output layer of the neural network is equal to the number of all available actions at a given state. After being transformed by a specific activation function, the value in each node represents the probability of each action being selected in the current state. The sum of the probabilities in all nodes is equal to 1.

2.2.2 Stochastic Policy Output in Continuous Action Domains

In continuous action domains, it is impossible for a node to represent an action; otherwise, an endless number of nodes would be required to represent all possible actions. Therefore, we cannot directly express actions produced from our policy (the probability distribution) via the output layer. Instead, we can generate a distribution from the output nodes and sample the desired actions from it. A distribution, such as the Gaussian distribution, does not model the probability of an action directly; rather, it models the chance of an action falling within a particular interval. The mean and the standard deviation are two primary parameters to construct a distribution; they are numbers that can be represented by two nodes in the output layer. The value of the mean is the action that the policy believes to be best, and the standard deviation is the level of confidence that the policy thinks we should have in that action (the larger the standard deviation, the lower the level of confidence).

2.3 Exploration and Exploitation

In continuous action domains, the learning environment is usually complex. Meanwhile, reinforcement learning is a method of completing a task through "trials and errors." As such, the agent needs to constantly try different actions to reach different environmental states and learn from these attempts. As a result, the best actions in different states are chosen, and these actions are combined to form a policy.

With the continuous update of the policies, the agent may have found a relatively good policy, but this policy is not the optimal solution for the task. Then, the agent needs to make a trade-off between trying new actions and continuing to use the actions from the existing policy. The process of an agent trying new actions to generate a new policy is called *exploration*, while the behavior of the agent continuing to use the actions from the existing policy is called *exploitation*. More specifically, exploration is to try more new actions at the cost of slightly deviating from the current best policy in an attempt to generate a better one. Exploitation aims to use the best-known policy to obtain higher cumulative rewards. In the learning process, the agent should not only make good use of the learned experience to obtain a higher reward as much as

possible but also keep exploring to try to find better policies constantly.

2.4 Decaying ϵ-greedy Method

In the actual learning process, the ratio of the agent's exploration and exploitation becomes the key to whether it can complete the learning efficiently. If the exploration range is extensive, the agent will continue to randomly attempt new actions without a goal. It is hard for the agent to use the proper actions to complete the tasks, and the training speed will be slowed. On the other hand, if the exploitation rate is too high, the agent will become "short-sighted" and its policy will fail to reach the optimal solution, thus failing to obtain a higher reward. Therefore, the ratio of exploration and exploitation will be an essential factor affecting the overall learning efficiency of the agent.

In order to define and reasonably allocate the exploration and exploitation ratio, the *ϵ-greedy* method is proposed. For each step, ϵ-greedy uses the probability ϵ as the probability of exploration and $1 - \epsilon$ as the probability of exploitation. More specifically, in ϵ-greedy, the agent will choose a random action with the probability of ϵ and the action under the current optimal policy with a probability of $1 - \epsilon$, and follow this rule to update the policies based on the performance of the new action combinations.

In practical scenarios, we usually think that with the advancement of the learning process, the policy learned by the agent will become closer to the optimal policy, and the reward value obtained by the policy will be higher. So, as learning progresses, we tend to gradually reduce the agent's exploration of new actions and focus more on the current policy to make the agent's learning more efficient. So, as the exploration rate gets lower and lower, the value of ϵ will gradually decrease. This method is known as *Decaying ϵ-greedy*.

2.5 Sampling and KL Divergence

In a discrete action domain, since the number of available actions is finite, the agent can directly select new actions by enumerating all the available actions. However, enumerating actions is not possible in a continuous action domain since the number of available actions is infinite. Therefore, to evaluate and select continuous actions, we need to sample a series of actions and evaluate the sampled actions as an approximation of the continuous actions. The actions after sampling will change from continuous to discrete, which can be integrated into the evaluation process; meanwhile, because of the reduced number of actions, sampling can significantly reduce the difficulty of calculation.

Although sampling can facilitate calculation, one problem brought by sampling cannot be ignored; there will be differences between the data distribution after sampling and the original data distribution. More

specifically, because the sampled data does not fully represent the original data, there will be a difference between the optimal policy learned by the agent after learning from the sampled data and the optimal policy using the original data. This difference may well result in the agent performing poorly on the task. The core factor in solving this problem will be how to minimize the gap between the policy learned by the agent using the sampled data and the actual optimal policy. It is conceivable that the main reason for the gap between policies through sampling is the loss of information. In this case, Kullback-Leibler (KL) divergence is proposed, which aims to measure the amount of information lost in the process of approximating another policy with one policy. KL divergence is calculated as the expectation of the *log* difference between the original action distribution p and the approximate action distribution q for a stochastic policy, where the *log* represents the number of bits of information expected to loss. The formula of KL divergence is:

$$D_{KL}(p\|q) = E[\log p(x) - \log q(x)] \tag{2.3}$$

More intuitively, we can understand p and q as the responding actual optimal policy and the policy learned by sampling, namely:

$$D_{KL}(\pi(s)_{optimal}\|\pi(s)_{approx}) = E[\log \pi(s)_{optimal} - \log \pi(s)_{approx}] \tag{2.4}$$

where s is the current state. It can be seen that the smaller the KL divergence, the closer the policy learned by the agent using the sampled data is to the actual optimal policy, and the better the agent will perform in the real tasks.

2.6 On-policy and Off-policy Update

For RL algorithms, one or more Q-functions are used to evaluate the potential future return of taking a specific action in a particular state. There are two ways for an RL agent to update its Q-function. One is called *on-policy update*, which updates the Q-functions directly with the data generated from the interaction with the environment, i.e., assuming the current policy will be followed after the current action to make decisions during the learning process. For example, *SARSA* [27] is a typical on-policy update method. In SARSA, the Q-function will be updated by using the Q-value of the next state and the action generated by the current policy. Hence, the return of the state-action pairs in SARSA will be evaluated by assuming the current policy continues to be followed. The update function for SARSA is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma Q\left(s', a'\right) - Q(s,a)\right) \tag{2.5}$$

where $Q(s,a)$ is the Q-value of the new state-action pair; r is the reward given by the environment; a' is the action that would be taken according to the current policy $\pi(s)$, and s' is the state after executing a'; γ is a discount factor, which can adjust the importance of the Q-value of the current state-action pair; α is the learning rate, which can adjust the importance of the environmental reward and the difference between the Q-value of the new action and the current action.

The other way the agent updates its Q-function is called *off-policy update*. In off-policy updates, such as in *Q-learning* [28], the Q-function is updated using the Q-value of the next state the action learned by the agent previously that gives the highest return. In Q-learning, the update function is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q\left(s', a'\right) - Q(s,a) \right) \tag{2.6}$$

where all the parameters are the same as in SARSA, and $Q(s', a')$ is the set of Q-values of all state-action pairs. Therefore, the return for state-action pairs in off-policy learning will be evaluated following a previously learned greedy approach.

2.7 Actor-Critic Architecture

The Actor-Critic architecture [2] is a commonly used architecture in deep reinforcement learning. This architecture contains two neural networks: the actor network and the critic network. The actor network is responsible for using learned policies to interact with the environment by performing different actions to obtain reward values; in other words, the actor network is a policy network, where the input is the current environmental state and the output is the corresponding action of the current policy. The critic network learns the reward mechanisms of the environment by observing the feedback received by the actor network and "critiquing" the actions of the actor network, where the input is the current state-action pair, and the output is the corresponding Q-value. The critic network can use the Q-values to help the agent judge the goodness of the current action and update the actor network; it can also learn the relationship between the environmental states and the rewards and see the expected rewards based on the agent's current state and action. The Actor-Critic architecture can usually enhance the learning efficiency of an agent. To illustrate, in a single neural network architecture, the agent must perform a policy update by completing the entire learning episode (e.g., a traditional policy gradient). In contrast, in Actor-Critic, with the help of the critic network, each action (step) will receive feedback from the critic network, thus enabling the agent to do single-step updates and reduce the cost of exploration.

Chapter 3

Related Work

3.1 Reward Machines, CRM and HRM

3.1.1 Reward Machines

When the agent searches for an optimal policy for a task, especially for a complex task, the agent needs to perform a large number of actions to judge the goodness of the current policy by the reward values from the environment. In traditional reinforcement learning, the agent does not know the complete environment model. Suppose the agent wants to complete a specified task. In that case, it must interact with the environment continuously by trying different actions and judging whether those actions are helpful to complete the task by the reward value given by the environment. In a complex task, there are usually a number of conditions that must be fulfilled in order to be completed. However, since the reward function is a black box, the agent cannot know the details of all the conditions, which will make the agent keep exploring blindly and thus waste lots of time in the exploration phase.

In order to help the agent enhance the exploration efficiency, Toro Icarte et al. proposed a finite state machine, called the *reward machines* [25, 24], which supports the specification of a wide range of non-Markovian rewards. In reward machines, the programmer can decompose a complex task into multiple subtasks by defining a reward machine state and reward function for each phase of a complex task. Each subtask can be defined by using the set of propositional symbols $\mathcal{P}$.

To have a simple and concrete example, suppose that the agent in Figure 3.1 is given the task of walking from point A to point D, and it will get a reward value of 1000 after reaching point D. The agent can move forward or backward on a line segment. Since the agent is far away from point D, if the task received by the agent consists only of reaching point D, the agent needs to spend a lot of time exploring. If a reward machine

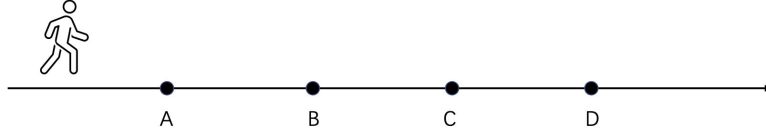

Figure 3.1: An example RM environment

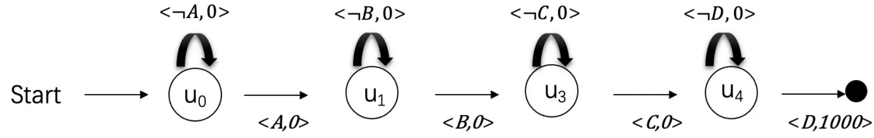

Figure 3.2: The automaton of the RM environment example

is used, the RM can decompose this task into several subtasks by adding intermediate target points for the agent to complete. Specifically, we can add several intermediate points A, B, and C between the agent's starting position and point D, so that the agent can first learn how to reach point A, then learn how to reach point B from point A, then learn how to reach point C from point B, and finally learn how to reach point D after reaching point C. In this way, the agent becomes closer to the target point in each subtask, which can reduce the time for the agent to explore, and thus the learning efficiency will be improved. The automaton for this task is shown in Figure 3.2. In this automaton, the reward value is 0 for transitions among the non-terminal RM states u_0 to u_4. When the agent reaches point D while in u_4, it arrives at the terminal RM state, and it will receive a reward value of 1000. In this environment, $\mathcal{P}$ can be defined as $\mathcal{P} = \{A, B, C, D\}$, where event $e \in \mathcal{P}$ occurs when the agent is at location e. To assign truth values to symbols in $\mathcal{P}$, a *labelling function* $L : S \times A \times S \to 2^{\mathcal{P}}$ will be needed. L can assign truth values to symbols in $\mathcal{P}$ given an environment experience (s, a, s'), where s' is the resulting state after executing action a from state s. The definition of a reward machine is: given a set of propositional symbols $\mathcal{P}$, a set of (environment) states S, and a set of actions A, a reward machine (RM) is a tuple $\mathcal{R}_{\mathcal{P}SA} = \langle U, u_0, F, \delta_u, \delta_r \rangle$ where U is a finite set of states, $u_0 \in U$ is the initial state, F is a finite set of terminal states (where $U \cap F = \emptyset$), δ_u is the state-transition function, $\delta_u : U \times 2^{\mathcal{P}} \to U \cup F$, and δ_r is the state-reward function, $\delta_r : U \to [S \times A \times S \to \mathbb{R}]$. In the example, U is the set of all the non-terminal RM states, including $\{u_0, u_1, u_2, u_3, u_4\}$; F is the set of the terminal state, which is the state after u_4; the state-transition function δ_u is used when the agent reaches point A, and it will transfer the agent's current RM state from u_0 to u_1 (otherwise it remains in u_0), then it will transfer the RM state from u_1 to u_2 when the agent reaches point B, and so forth. When the agent reaches point D, indicating that it has reached the terminal state, the state-reward function δ_r is used and it will give the agent a reward value of 1000.

A reward machine can take abstracted environment descriptions as input, and produce reward functions as output. In this way, the reward function is no longer a black box, and the learning algorithm can exploit knowledge of the reward machine structure. As a result, each subtask can pass its high-level requirements to the agent. The agent will be able to explore according to the specific instructions given by the subtasks. By completing each subtask sequentially, the agent will be able to complete a complex task and achieve the final goal eventually.

3.1.2 Counterfactual Experience for reward machines (CRM)

In general reinforcement learning, the underlying environment model of the agent is assumed to be a *Markov Decision Process* (MDP). An MDP [6] is a tuple $\mathcal{M} = \langle S, A, r, p, \gamma, \mu \rangle$, where S is a finite set of states, A is a finite set of actions, $r : S \times A \times S \rightarrow \mathbb{R}$ is the reward function, $p\left(s_{t+1} \mid s_t, a_t\right)$ is the transition probability distribution, $\gamma \in (0, 1]$ is the discount factor, and μ is the initial state distribution where $\mu\left(s_0\right)$ is the probability that the agent starts in state $s_0 \in S$. By using reward machines, the agent learns in the environment considering not only the state s_t at time t of the environment, but also the RM state u_t at time t. The extra consideration of the RM state u_t changes the learning environment from a traditional MDP to a *Markov Decision Process with a Reward Machine (MDPRM)* [23]. A *Markov Decision Process with a Reward Machine (MDPRM)* is a tuple $\mathcal{T} = \langle S, A, p, \gamma, \mu, \mathcal{P}, L, U, u_0, F, \delta_u, \delta_r \rangle$, where S, A, p, γ and μ are defined as in an MDP, $\mathcal{P}$ is a set of propositional symbols, L is a labelling function $L : S \times A \times S \rightarrow 2^{\mathcal{P}}$, and U, u_0, F, δ_u, and δ_r are defined as in a reward machine. In an MDPRM, the policy learned by the agent then changes from $\pi(a \mid s)$ to $\pi(a \mid s, u)$, and the learning experience generated by the agent changes from $\langle s, a, r, s' \rangle$ to $\langle s, u, a, r, s', u' \rangle$. It can be seen that MDPRMs are regular MDPs when considering the cross-product between the environment states S and the RM states U. As such, the standard RL algorithms that learn in MDPRMs use the *cross-product baseline* of reward machines.

As a concrete example, Algorithm 1 shows the pseudocode for using the cross-product baseline in Q-learning. In the cross-product Q-learning, besides considering the environment states s, it also keeps track of the current RM state u and learns the Q-values over the cross-product $Q(s, u, a)$. This lets the agent consider the current environment state s and RM state u when selecting the following action a. On line 8, the Q-value is updated as follows:

$$Q(s, u, a) \stackrel{\alpha}{\longleftarrow} r + \gamma \max_{a' \in A} Q\left(s', u', a'\right) \tag{3.1}$$

where γ is the learning rate, which is the same as in traditional Q-learning, and r is the reward given by the

reward machine. This uses the $\stackrel{\alpha}{\longleftarrow}$ notation, which is an abbreviation for:

$$Q(s,u,a) \leftarrow \alpha * (r + \gamma \max_{a' \in A} Q(s',u',a')) + (1-\alpha) * Q(s,u,a) \tag{3.2}$$

where α is the (learning) rate to update the Q-value. However, the RL algorithms using the cross-product baseline do not exploit the information exposed by the RM.

Algorithm 1 The cross-product baseline using Q-learning.

Input: $S, A, \mu, \gamma \in (0,1], \alpha \in (0,1], \epsilon \in (0,1], \mathcal{P}, L, \mathcal{R}_{\mathcal{P}SA} = \langle U, u_0, F, \delta_u, \delta_r \rangle$

1: Initialize $Q(s,u,a)$ for all $s \in S, u \in U \cup F$, and $a \in A$, arbitrarily except that $Q(s',u',\cdot) = 0$ if s' is terminal or $u' \in F$
2: **for** however many updates **do**
3: Initialize $u \leftarrow u_0$ and sample an initial state for the environment $s \sim \mu$
4: **while** s is not terminal and $u \notin F$ **do**
5: Choose action a from (s,u) using policy derived from Q(e.g., ϵ-greedy)
6: Take action a and observe the next state s'
7: Compute the reward $r \leftarrow \delta_r(u)(s,a,s')$ and next RM state $u' \leftarrow \delta_u(u, L(s,a,s'))$
8: $Q(s,u,a) \stackrel{\alpha}{\longleftarrow} r + \gamma \max_{a' \in A} Q(s',u',a')$
9: Update $s \leftarrow s'$ and $u \leftarrow u'$
10: **end while**
11: **end for**

To exploit the information provided by the RM, Toro Icarte et al. proposed a method called *Counterfactual experience for Reward Machines (CRM)* [23]. CRM also learns policies over the cross-product $\pi(a \mid s,u)$, but uses counterfactual reasoning to generate *synthetic* experiences. In CRM, the RM will go through every RM state $\bar{u} \in U$ after each action performed by the agent, and use the state transition function $\delta_u(\bar{u}, L(s,a,s'))$ to determine the next RM state $\bar{u}'$; the agent will also receive a reward of $\bar{r}$ using the reward transition function $\delta_r(\bar{u})(s,a,s')$. That is, instead of just providing the actual experience in an MDPRM, the RM can now provides one experience per RM state, and the agent will now also learn the experience provided by the state-transition function and state-reward function.

CRM allows the agent to learn in parallel. During the learning process using CRM, the agent not only learns the actual experience in MDPRM, but the counterfactual experiences are also continuously being stored. Whenever the agent reaches an RM state that it has reached before, the agent will be able to use the previous learned counterfactual experience in this state to learn more efficiently. For example, suppose the agent's task is to depart from the workplace to the post office and then go home (the workplace, the post office, and the home are in different locations). The reward machine will have two RM states or two subtasks in this environment. The first subtask is to go from the workplace to the post office, and the second subtask is to go from the post office to home. While the agent is exploring, it may arrive home without going through the post office. In this case, CRM will use what the agent has learned from the workplace's direct-to-home

experience to determine that this behavior is not the agent's first task, and it is not an efficient behavior to go to the post office. However, at the same time, the CRM has also learned part of the experience of how to get home. Therefore, in the following exploration, once the agent has completed the subtask of going from the workplace to the post office, the agent will be able to use the part of the experience that the CRM has learned before to complete the second subtask more quickly.

CRM can be combined with some reinforcement learning algorithms to help the original algorithms achieve better training results because it lets the agent exploit the information from reward machines. First, the agent learns in an MDPRM, makes an action a in the cross-product state $\langle s, u\rangle$, and then reaches the next cross-product state $\langle s', u'\rangle$ and receives the reward value r. At this point, the agent's judgment of the environment state will not only refer to the environment states s and s' alone, but will also consider the RM states u and u'. The actual experience received by the agent in CRM is (s, u, a, r, s', u'). Moreover, the agent will also learn the state and reward transitions in CRM, which can give the agent key information about how to transfer from one RM state to another. The comprehensive consideration of the environment state and the RM state can help the agent determine its task process better and thus improve the policy. For example, the combination of CRM and Q-learning [28] adds the counterfactual experience generated by CRM when computing the Q-value.

Algorithm 2 shows the pseudocode of Q-learning with CRM, which includes the learning process using the actual experience and the counterfactual experience in MDPRM.

Algorithm 2 Q-learning with counterfactual experiences for RMs (CRM).

Input: $S, A, \mu, \gamma \in (0,1], \alpha \in (0,1], \epsilon \in (0,1], \mathcal{P}, L, \mathcal{R}_{\mathcal{P}SA} = \langle U, u_0, F, \delta_u, \delta_r\rangle$

1: Initialize $Q(s, u, a)$ for all $s \in S, u \in U \cup F$, and $a \in A$
2: **for** however many updates **do**
3:     Initialize $u \leftarrow u_0$ and sample an initial state for the environment $s \sim \mu$
4:     **while** s is not terminal and $u \notin F$ **do**
5:         Choose action a from (s, u) using policy derived from Q(e.g., ϵ-greedy)
6:         Take action a and observe the next state s'
7:         Compute the reward $r \leftarrow \delta_r(u)\left(s, a, s'\right)$ and next RM state $u' \leftarrow \delta_u\left(u, L\left(s, a, s'\right)\right)$
8:         Set experience $\leftarrow \{\left(s, \bar{u}, a, \delta_r(\bar{u})\left(s, a, s'\right), s', \delta_u\left(\bar{u}, L\left(s, a, s'\right)\right)\right) \mid \forall \bar{u} \in U\}$
9:         **for** $(s, \bar{u}, a, \bar{r}, s', \bar{u}') \in$ experience **do**
10:             $Q(s, \bar{u}, a) \xleftarrow{\alpha} r + \gamma \max_{a' \in A} Q\left(s', \bar{u}', a'\right)$
11:         **end for**
12:         Update $s \leftarrow s'$ and $u \leftarrow u'$
13:     **end while**
14: **end for**

In the pseudocode, the generation of the counterfactual experience occurs in line 8. After the agent executes an action, for each RM state, CRM will use the state-transition function to calculate the next RM state and use the reward-state function to calculate the reward. Then, CRM stores the counterfactual experience of each RM state in *experience*. That is, if there are ten RM states in the whole task, there will

be ten counterfactual experiences added in *experience* for each action, and each counterfactual experience corresponds to one RM state.

With the help of the counterfactual experience, the agent can judge whether it has accomplished one specific subtask defined by reward machines. To illustrate, multiple subtasks will be created when the programmer defines a complex task using reward machines. When the agent learns the given task, it will start from the initial RM state, which can be seen as the first subtask. In CRM, after one action, the agent will receive the next RM state $\bar{u}'$. According to $\bar{u}'$, the agent will tell whether the next RM state or, more specifically, the next subtask has begun. Suppose the next RM state is still the same as the current RM state. In that case, the agent did not accomplish the current subtask, and it still needs to stay at the current RM state to do more exploration. However, if the next RM state changes, it indicates the agent has completed the current subtask and will move to the next RM state, then start trying to complete the next subtask.

For the reward part, the agent now refers to reward $\bar{r}$ given by the reward machines. The value of the reward will be determined by the current environmental state s, the action a executed by the agent, the next environmental state s', and the current reward machine state u. The new reward will be represented as $\bar{r} \leftarrow \delta_r(u)\,(s, a, s')$, where $\delta_r : U \rightarrow [S \times A \times S \rightarrow \mathbb{R}]$ is the state-reward function. The reward given by the RM can help the agent judge the correctness of its actions and tell the agent its progress on the given task.

In summary, CRM's responsibility is to store related information causing the change of the RM states. Later, the agent can re-invoke this information when it encounters the same state so that the agent knows what to do to move from the current RM state to the next RM state, thus completing the whole task faster. Note that in CRM, a change in the environment state does not necessarily lead to a change in the RM state. Suppose a change in S does not cause a change in U. In that case, this tells the agent that the current action may not be helpful in completing the current subtask. In the later training, the agent will reduce the probability of occurrence of the same action in the same state, which will encourage the agent to explore other suitable actions and thus accelerate the learning efficiency. However, once the environment state change causes the RM state's change, when the agent reaches the same environment state again, the agent can use the related experience to tell what action to perform to get to the next RM state; thus, the agent can do less exploration in that specific state, which can significantly enhance the learning efficiency.

3.1.3 Hierarchical Reinforcement Learning for Reward Machines (HRM)

There is another use case for reward machines, called Hierarchical Reinforcement Learning for Reward Machines (HRM) [23]. This approach is based on the options framework for *hierarchical reinforcement learning* (HRL) [20] . The general goal of HRM is to break down a complex problem into simpler subproblems

that are potentially easier to solve. Formally, an option is a triple $\langle \mathcal{I}, \pi, \beta \rangle$, where $\mathcal{I}$ is the initiation set (the subset of the state space of the RM in which the option can be started), π is the policy that determines what actions should be taken while the option is being pursued, and β gives the probability that the option will terminate in each state. It is worth noting that an option can be viewed as a macro-action.

In HRM, the agent will learn a set of options that focus on learning how to shift from one RM state to another RM state in an MDPRM. Then, to receive the reward, a higher-level policy will learn how to choose among those options.

For example, suppose the given task for the agent is to pick up a coffee and the mail, then deliver them to the office. The agent has two ways to complete this task; it can either pick up the coffee and then the mail, or pick up the mail first and then the coffee, and eventually head to the office. In this case, HRM will learn a total of five options. Specifically, HRM will learn one policy to get a coffee before getting the mail, one policy to get the mail before getting a coffee, one policy to get a coffee after getting the mail, one policy to get the mail after getting the coffee, and one policy to go to the office. The high-level policy's job is to determine which of the five options to execute next. The high-level policy, for example, will decide whether to grab a coffee or the mail first after the task is initialized. It will assess the current state of the environment when making the decision, and it will learn to get the coffee or the mail based on which is closer to the agent.

HRM can be particularly effective at quickly learning good policies for MDPRMs, according to the experimental results by Toro Icarte et al. Its success comes from its capability to learn policies for all options in parallel using off-policy learning. However, HRM may lead to sub-optimal solutions. The reason is that the options-based approach is greedy; the policies will always strive to transition as rapidly as possible without considering the subsequent performance after the transition. An option's policy only tries to reach its termination region without considering what needs to be done afterwards; it may not reach the best state in the termination region for the subsequent subtask.

3.2 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) [15] was proposed by Lillicrap et al. in 2016, and it is an off-policy deep RL algorithm using the actor-critic architecture. DDPG can solve continuous control problems by using an actor network to learn the optimal policy and a critic network to approximate the optimal Q-function. The input of the actor network is the current environmental state, and the output is a (possibly continuous-type) action to execute derived from the current policy. Specifically, the actor network is a policy

function with the expression:

$$\mu_\theta(s) : S \rightarrow A \tag{3.3}$$

where μ_θ is the agent's current policy, which can derive an action based on the current environmental state; and θ is the parameters of the deep neural network that represents the policy. For example, in Half-Cheetah [3], the input of the actor network is a dictionary type of array with a length of 17, which includes the coordinate position of the robot, the angles of the limbs, and the velocities of the joints. The range of the array elements is $(-\infty, \infty)$. Correspondingly, the output is an action; in Half-Cheetah, it is a dictionary type of array with a length of 6, including the torque that will be applied to each joint. Each element in the array is a 32-bit float number with a range of $(-1, 1)$, which represents the amount of force that will be applied.

In comparison, the input of the critic network is the current state-action pair, and the output is the corresponding Q-value. For example, in Half-Cheetah, the input is a two-dimensional array that combines all the state and action information in the actor network. Specifically, the critic network is a value function with the expression:

$$Q_\phi(s, a) : S \times A \rightarrow \mathbb{R} \tag{3.4}$$

where ϕ is the parameters of the deep neural network that represents the Q-function. The Q-function is updated by minimizing the mean squared Bellman error (MSBE) between $Q_\phi(s, a)$ and $r + Q_{\phi_{targ}}\left(s', \mu_{\theta_{targ}}(s')\right)$, where ϕ_{targ} and θ_{targ} are the parameters of target networks to evaluate the Q-values in the critic network and the policies in the actor network. The output Q-value is a real number, which represents the expected reward after the agent apply an action a in state s, and do the remaining actions by following μ:

$$Q_\phi(s, a) = \underset{s_t, a_t \sim \pi_\mu}{\mathbb{E}}\left[\sum_{t=0}^{\infty} \gamma^t r\left(s_t, a_t\right) \mid s_0 = s, a_0 = a\right] \tag{3.5}$$

where π_μ represents the distribution of the state-action pairs under the policy μ. $Q_\phi(s, a)$ can be seen as a score for a policy under state s and action a.

The learning method of DDPG is similar to Q-learning in that both learn to derive the optimal action value function:

$$Q^*(s, a) = \underset{s' \sim P}{\mathbb{E}}\left[r(s, a) + \gamma \max_{a'} Q^*\left(s', a'\right)\right] \tag{3.6}$$

where $s' \sim P$ is the next state sampled by the environment from a distribution $P(\cdot \mid s, a)$. Unlike in standard Q-learning, because of the continuous action space, the number of actions is infinite, so it is very costly to analyze each action and update the Q-function. For this reason, in DDPG, the original Q-function

is compiled into a differentiable form, where the action a is approximated as $\mu(s)$, and the experience $(s, \mu(s), r, s')$ learned by the agent is divided into multiple mini-batches to update the Q-function in stages, thus obtaining an approximation of the Q-function. Then the policy in the actor network is updated using the expected gradient [1] of $Q^*(s, a)$, and the parameters of the target networks are periodically updated as follows: $\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$ and $\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$, where $\rho \in (0,1)$.

3.3 DDPG with CRM

In traditional DDPG, the agent will receive feedback from the environment after making an action according to the current policy $\pi(a \mid s)$, including the current state s, the reward value r, and the next state s'. Then, the agent stores the feedback information from the environment and the actions it takes in the replay buffer in the form of tuples, expressed as $\langle s, a, r, s' \rangle$.

To improve the learning efficiency of DDPG, especially the learning capability in complex tasks, DDPG-CRM was proposed by Toro Icarte et al. [23], and a series of experiments show the significant performance improvement after adding CRM. However, DDPG-CRM was not presented in detail by Toro Icarte et al., so a brief analysis of DDPG-CRM with pseudocode will be given in this section.

After combining CRM with DDPG, the learning environment will no longer be a traditional MDP but an MDPRM. First, unlike the original DDPG, since the learning environment has become an MDPRM, the information received by the agent will include the data of the environment in addition to the information provided by the reward machines. The data of reward machines includes the current RM state u and the next RM state u'. The actual experience now becomes to $\langle s, u, a, r, s', u' \rangle$. Moreover, CRM will also generate one counterfactual experience for each RM state $\bar{u} \in U$, and the agent will also learn the experience provided by the state-transition function $\bar{u}' \leftarrow \delta_u(\bar{u}, L(s, a, s'))$ and state-reward function $\bar{r} \leftarrow \delta_r(\bar{u})(s, a, s')$. Because of the addition of the RM information, the experience received by the agent will also include the current RM state $\bar{u}$, RM reward $\bar{r}$ and next RM state $\bar{u}'$. The data stored in the replay buffer now includes counterfactual experiences and becomes a new form of synthetic experience: $\{\langle s, \bar{u}, a, \delta_r(\bar{u})(s, a, s'), s', \delta_u(\bar{u}, L(s, a, s')) \rangle \mid \bar{u} \in U\}$.

At this time, the policy update of DDPG-CRM will change from considering the environment states and rewards to a comprehensive consideration of the environment states, RM states, and the reward given by reward machines. Similarly, the critic network of DDPG-CRM will additionally refer to the RM states and the RM reward based on the original environment information when updating the Q-function. The pseudocode of DDPG-CRM is shown in Algorithm 3, where the counterfactual experiences are generated in line 7. Then, the RM states and rewards will be used to generate the target Q-function (line 13), and the Q-

function will be updated by minimizing the Mean Squared Bellman Error (MSBE) with stochastic gradient descent (line 14), where ∇_{ϕ_i} represents the gradient that is approximated using sampled mini-batches from the replay buffer *experience*. Based on the Q-values of the current state-action pair, the corresponding policy will be updated (lines 15 and 16).

Algorithm 3 DDPG with counterfactual experiences for RMs (CRM).

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$, labelling function L, a finite set of states U, a finite set of terminal states F, state-transition function δ_u, state-reward function δ_r, initial RM state $u_0 \in U$

1: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta, \phi_{\text{targ}} \leftarrow \phi$
2: Initialize $u \leftarrow u_0$ and $s \leftarrow$ EnvInitialState()
3: **repeat**
4: Observe state s, u and select action $a = \text{clip}\left(\mu_\theta(s,u) + \epsilon, a_{\text{Low}}, a_{\text{High}}\right)$, where $\epsilon \sim \mathcal{N}$
5: Execute a in the environment and observe next state s'
6: Compute the reward $r \leftarrow \delta_r(u)(s,a,s')$ and next RM state $u' \leftarrow \delta_u(u, L(s,a,s'))$, done signal d to indicate whether s' is terminal, and check whether $u \in F$
7: Set experience $\leftarrow \{(s, \bar{u}, a, \delta_r(\bar{u})(s,a,s'), s', \delta_u(\bar{u}, L(s,a,s')), d) \mid \bar{u} \in U\}$
8: Store experience in replay buffer $\mathcal{D}$
9: If s' is terminal or $\bar{u} \in F$, reset environment state.
10: **if** it's time to update **then**
11: **for** however many updates **do**
12: Randomly sample a batch B of transitions from $\mathcal{D}$
13: Compute targets:

$$y(\bar{r}, s', \bar{u}', d) = \bar{r} + \gamma(1-d) Q_{\phi_{\text{targ}}}\left(s', \bar{u}', \mu_{\theta_{\text{targ}}}(s', \bar{u}')\right)$$

14: Update Q-functions by minimizing the MSBE of the sampled Q-function and target Q-function using one step of gradient descent

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,\bar{u},a,\bar{r},s',\bar{u}',d)\in B} \left(Q_{\phi_i}(s,\bar{u},a) - y(\bar{r}, s', \bar{u}', d)\right)^2$$

15: Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s,\bar{u}\in B} Q_\phi\left(s, \bar{u}, \mu_\theta(s,\bar{u})\right)$$

16: Update target network with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$$

17: **end for**
18: **end if**
19: **until** convergence or maximum training step reached

3.4 Maximum Entropy Reinforcement Learning and Soft Actor-Critic (SAC)

3.4.1 Maximum Entropy Reinforcement Learning (MERL)

Entropy can be interpreted as the degree of chaos, disorder, and randomness. The higher the entropy, the richer environmental information will be contained by the environment. The benefits of introducing entropy into the RL algorithm are that the policy can be as random as possible. The agent can explore the state space more thoroughly and avoid the policy falling into a local optimum early. Multiple feasible solutions can be explored to complete the specified task, improving the resistance to interference by the environmental noise.

To calculate the entropy value of the random variable x, assume that x belongs to the distribution P. The entropy value of x, which is $H(P)$, is:

$$H(P) = \underset{x \sim P}{\mathrm{E}}[-\log P(x)] \tag{3.7}$$

In reinforcement learning, the agent's goal is to find the policy that collects the highest cumulative rewards, so the expression for the policy is:

$$\pi_{std}^{*} = \arg\max_{\pi} \sum_{t} \mathbb{E}_{(s_t, a_t) \sim \rho_\pi} \left[r\left(s_t, a_t\right) \right] \tag{3.8}$$

where ρ_π represents the distribution of the state-action pairs under policy π. Now, if the mechanism of entropy maximization is introduced into the search for policies, the target policy of the agent will become:

$$\pi_{\mathrm{MaxEnt}}^{*} = \arg\max_{\pi} \sum_{t} \mathbb{E}_{(s_t, a_t) \sim \rho_\pi} \left[r\left(s_t, a_t\right) + \alpha H\left(\pi\left(\cdot \mid s_t\right)\right) \right] \tag{3.9}$$

where α is the hyperparameter named temperature coefficient, which is a non-negative real number that is used to adjust the importance of the entropy value.

From the new policy formulation with the addition of maximum entropy, it can be seen that the learning objective of the policy will maximize the entropy value of the policy while maximizing the cumulative reward. The increase of the entropy value of the policy can help the agent to explore within a reasonable range as much as possible while ensuring that the policy has the highest possible reward value, avoiding letting the agent fall into the local optimum too early and laying a good foundation for the subsequent exploitation.

The algorithms based on MERL have a unique action exploration mechanism. In other mainstream deep

reinforcement learning algorithms (such as Q-learning and DQN), the decaying ϵ - greedy approach is their default method of controlling exploration probabilities. The decaying ϵ - greedy's approach is straightforward, that is, by defining a hyperparameter ϵ as the exploration probability of the entire learning process and gradually reducing the value of ϵ as the learning progresses to reduce exploration. In comparison, algorithms that learn by adding entropy (such as Soft Actor-Critic) do not randomly explore new actions with a fixed probability, but dynamically adjust the appearance probabilities of actions based on their Q-values. These probabilities include both actions that have been explored and new actions that have not yet been explored. In general, decaying ϵ - greedy is more inclined to randomly explore new actions with a fixed probability, while maximum entropy learning is more focused on learning to continuously assign the probability of each action. Moreover, because of the addition of entropy, those actions that seem to be unable to achieve a high Q-value but have a high entropy value will also be fully considered, and their probabilities of being selected can even be almost the same as the actions with the highest Q-value but low entropy value. This mechanism ensures that all the potential excellent actions can be reasonably allocated and utilized.

3.4.2 Soft Actor-Critic (SAC)

The Soft Actor-Critic (SAC) algorithm [10] was proposed by Haarnoja et al., and a series of benchmark environments confirmed its excellent performance. Later in 2018, Haarnoja et al. went a step further and applied SAC to real-life robot control tasks and achieved good results [11]. More than other RL algorithms, SAC adds an entropy term in the Q-function to maximize the entropy of the policy while maximizing the cumulative return. The benefit of the maximum entropy model is that the model makes the fewest assumptions about the unknown information of the environment when matching the observed information. Moreover, controlling the entropy value allows the agent to maintain a high level of exploration capability and prevent the agent from falling into a local optimum prematurely.

Similar to the structure of DDPG, SAC also uses the actor-critic architecture. The actor network is responsible for iterating the policy by receiving the information of the interaction between the agent and the environment. The critic network is responsible for updating the value function (Q-function) to judge the policy.

3.4.2.1 Soft Value Function (SVF) in critic network

Because of the addition of maximum entropy learning, SAC has a unique expression of the value function. The value function updated by the critic network in SAC refers to the expressions of the standard Q function. On this basis, the parameters required for maximum entropy learning are introduced. This new form of value

function is called Soft Value Function (SVF). The original standard Q-function is:

$$Q^{\pi}(s,a) = \underset{s_t,a_t\sim\rho_\pi}{\mathbb{E}}\left[\sum_{t=0}^{\infty}\gamma^t r\left(s_t,a_t\right) \mid s_0=s, a_0=a\right] \tag{3.10}$$

By introducing the target function of MERL π^*_{MaxEnt} (3.9), the Soft Value Q-function can be obtained as:

$$Q^{\pi}_{soft}(s,a) = \underset{s_t,a_t\sim\rho_\pi}{\mathbb{E}}\left[\sum_{t=0}^{\infty}\gamma^t r\left(s_t,a_t\right) + \alpha\sum_{t=1}^{\infty}\gamma^t H\left(\pi\left(\cdot \mid s_t\right)\right) \mid s_0=s, a_0=a\right] \tag{3.11}$$

It can be seen that after adding the entropy term to the Q-function, the reward value that the current policy obtains will no longer be the only criterion to judge the goodness of a policy. Instead, each update step of the Q-function receives a bonus entropy value. At this point, if the Q-value is high, it not only indicates that the reward value of the current policy is high but also that the agent is still maintaining a high level of exploration.

Specifically, a higher Q-value with an added entropy term may indicate three different cases. The first case is the most desirable; both the reward and entropy values are high, indicating that the agent's current policy is good; meanwhile, the agent also maintains a high level of exploration. The second case is that the reward value is high, but the low entropy value. This situation indicates that although the agent's policy seems to be good, its exploration capability has been lower, and the policy may have fallen into a local optimal solution, which is not a good thing for later policy improvement. The third case is that the current policy has a low reward value but a high entropy value. It indicates that although the current policy is not excellent, the agent still maintains a high exploratory nature and is constantly trying new actions, which means the policy has a higher potential to be improved. So it can be seen that when we use the Q-value after adding the entropy value as the criterion to judge a policy, we tend to choose not only the policies with high reward value but also policies with more random choices and high exploratory nature, even if the reward value they can obtain right now is not that outstanding, which is the core of SVF. SVF allows agents to open their minds to think about policies critically; when judging a policy, a high reward value does not mean it is an excellent policy, and a low reward value does not mean that the policy is worthless.

3.4.2.2 Energy Based Policy(EBP) in actor network

In order to adapt to more complex tasks and enhance the expressiveness of policies, SAC adopts a unique policy model. The policy distribution in SAC abandons the traditional Gaussian distribution and uses an

energy-based model to represent the policy:

$$\pi\left(a_t \mid s_t\right) \propto \exp\left(-\mathcal{E}\left(s_t, a_t\right)\right) \tag{3.12}$$

where $\mathcal{E}$ is the energy function, which represents the distribution characteristics of the Q-value, expressed as:

$$\mathcal{E}\left(s_t, a_t\right) = -\frac{1}{\alpha} Q_{soft}\left(s_t, a_t\right) \tag{3.13}$$

Based on the energy function, we can obtain the policy model as:

$$\pi\left(a_t \mid s_t\right) \propto \exp\left(\frac{1}{\alpha} Q_{soft}\left(s_t, a_t\right)\right) \tag{3.14}$$

Compared with the traditional Gaussian distribution model, the policy model based on energy function distribution has one prominent feature, which enhances the policy's exploratory nature. Figure 3.3 [21]

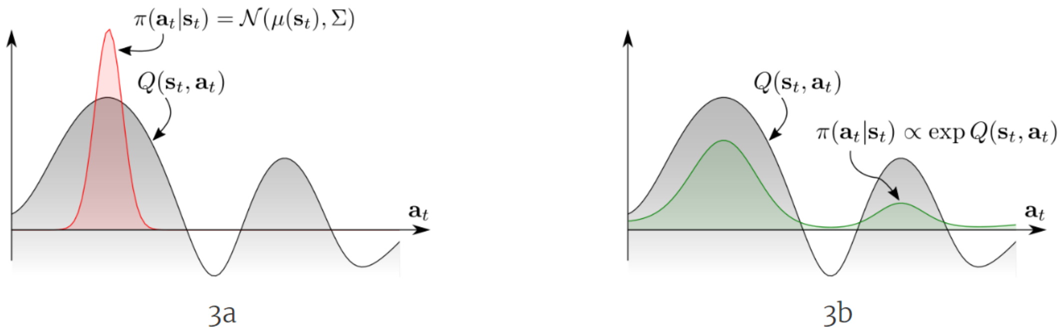

Figure 3.3: A multimodal Q-function

shows a typical distribution of a multimodal Q-function. In the state s_t, all available actions a_t of the agent constitute the horizontal axis, and their corresponding Q-values constitute the curve. It can be seen that there are two peaks in the curve, which respectively represent the optimal action interval and the sub-optimal action interval in s_t.

In the Gaussian distribution used by most RL algorithms, the policy's choice of actions will focus on the region with the largest Q-value (red distribution in 3a) and completely ignore all sub-optimal solutions. However, in the practical learning process, the sub-optimal solution is also valuable; in fact, the optimal solution in the current state is often not the global optimal solution. Therefore, when choosing an action, an ideal scenario is that the policy takes the optimal actions as the primary reference object, but also considers the sub-optimal actions. 3b is the case where the optimal solution and the sub-optimal solution

are considered at the same time. In 3b, the policy's choice of actions (green distribution in 3b) is no longer concentrated around the optimal solution. Instead, it assigns a part of the weight to the sub-optimal solution while mainly considering the optimal solution. In this way, the policy's selection of actions will no longer focus on the optimal action only, but will also fully consider and try the sub-optimal actions. Therefore, EBP uses the policy distribution in 3b to enhance its exploration ability.

3.4.2.3 Policy Evaluation and Improvement in SAC

SAC follows the operation mode of the actor-critic structure by alternately running policy evaluation and policy improvement to make the value function and the policy it guides continuously approach the optimal solution. Eventually, a good enough policy can be obtained and used in practical applications.

For policy evaluation part in the critic network, SAC uses the soft Q-function with the entropy term to iterate, which is shown in equation (3.11).

For the policy improvement part in the actor network, SAC uses a temporary policy π to interact with the environment. Then, it makes π as close to the energy based policy (EBP) as possible in subsequent policy optimization. The distance between π and EBP is measured by KL-divergence. The policy optimization formula is:

$$\pi_{new} = \arg\min_{\pi \in \Pi} D_{KL}\left(\pi\left(\cdot \mid s_t\right) \| \frac{\exp\left(\frac{1}{\alpha} Q_{\text{soft}}^{\pi_{\text{old}}}\left(s_t, \cdot\right)\right)}{Z_{\text{soft}}^{\pi_{\text{old}}}\left(s_t\right)}\right) \tag{3.15}$$

where Π represents the set of available policies, Z represents the expectation of the cumulative return in each state using the current policy.

Finally, similar to the common actor-critic type RL algorithms, SAC will alternately execute the policy evaluation and improvement to converge to the optimal value function and policy.

3.4.2.4 Implementation of SAC

The SAC algorithm adopts the actor-critic network structure, including one actor network and one critic network. The actor network is responsible for the policy update, and the critic network is responsible for updating the value function. The policy in the actor network is $\pi_\phi(\cdot \mid s)$, which represents the policy that the agent will take in the state s. The Q-function of the critic network is $Q_\theta(s, a)$, which represents the estimated Q-value obtained by the agent when taking action a in the state s.

First, the critic network will receive an agent's state-action pair (s, a) and output a real value as the Q-value for this state-action pair. Then, the actor network will receive the state s that the agent is in, then output a mean value and a standard deviation to form a distribution of available actions in s. When the agent needs to select an exact action from the policy, it will sample an action from the distribution and use

it as the next action.

Because all the actions are sampled from a distribution, there will be loss of information, thus causing a difference between the expected Q-values and the Q-values of the sampled state-action pairs. In order to make the Q-values of the sampled state-action pairs as close to the expected Q-values as possible, a loss function that can measure their difference is needed. The loss function can be defined as the difference between the Q-values of the sampled state-action pairs and the actual state-action pairs. According to equation (3.11), the loss function for training the Q-function can be obtained:

$$J_Q(\theta) = \underset{\substack{(s_t,a_t,s_{t+1})\sim\mathcal{D} \\ a_{t+1}\sim\pi_\phi}}{\mathbb{E}} \left[\frac{1}{2}\left(Q_\theta\left(s_t, a_t\right) - \left(r\left(s_t, a_t\right) + \gamma\left(Q_\theta\left(s_{t+1}, a_{t+1}\right) - \alpha \log\left(\pi_\phi\left(a_{t+1} \mid s_{t+1}\right)\right)\right)\right)\right)^2\right] \tag{3.16}$$

where γ and α are discount factors, and (s_t, a_t) is extracted from the data generated from the replay buffer that stores the past interactions between the agent and the environment, and a_{t+1} is collected from the temporary policy π_ϕ.

Moreover, since there is a difference between the expected Q-values and the Q-values of the sampled state-action pairs, there will be a difference between the expected optimal policy and the policy based on the sampled data. As such, a loss function that can measure the difference between the expected optimal policy and the temporary policy is needed. KL-divergence can be used to measure the difference between policies. Similar to equation (3.15) that we used to update a policy, the loss function of the temporary policy π_ϕ can be obtained as:

$$\begin{aligned}
J_\pi(\phi) &= D_{\mathrm{KL}}\left(\pi_\phi\left(\cdot \mid s_t\right) \| \exp\left(\frac{1}{\alpha} Q_\theta\left(s_t, \cdot\right) - \log Z\left(s_t\right)\right)\right) \\
&= \mathbb{E}_{s_t\sim\mathcal{D}, a_t\sim\pi_\phi}\left[\log\left(\frac{\pi_\phi\left(a_t \mid s_t\right)}{\exp\left(\frac{1}{\alpha} Q_\theta\left(s_t, a_t\right) - \log Z\left(s_t\right)\right)}\right)\right] \\
&= \mathbb{E}_{s_t\sim\mathcal{D}, a_t\sim\pi_\phi}\left[\log \pi_\phi\left(a_t \mid s_t\right) - \frac{1}{\alpha} Q_\theta\left(s_t, a_t\right) + \log Z\left(s_t\right)\right]
\end{aligned} \tag{3.17}$$

where α is a discount factor, s_t is taken from the replay buffer, and a_t is sampled from the current temporary policy π_ϕ.

After that, the agent will continuously collect data by interacting with the environment, train the two networks, and try to minimize the value of two loss functions $J_Q(\theta)$ and $J_\pi(\phi)$, which eventually converges to a solution. The pseudocode of SAC is shown in Algorithm 4, where line 11 calculates the target Q-function, line 12 updates the Q-function by minimizing the MSBE between the sampled Q-function and the target Q-function ($J_Q(\theta)$), and lines 13 and 14 update the policy by minimizing the difference between the temporary policy and the target optimal policy ($J_\pi(\phi)$).

Algorithm 4 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$

1: Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1, \phi_{\text{targ},2} \leftarrow \phi_2$
2: **repeat**
3: Observe state s and select action $a \sim \pi_\theta(\cdot \mid s)$
4: Execute a in the environment
5: Observe next state s', reward r, and done signal d to indicate whether s' is terminal
6: Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
7: If s' is terminal, reset environment state.
8: **if** it's time to update **then**
9: **for** j in range (however many updates) **do**
10: Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
11: Compute targets for the Q functions:

$$y\left(r, s', d\right) = r + \gamma(1-d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}\left(s', \tilde{a}'\right) - \alpha \log \pi_\theta\left(\tilde{a}' \mid s'\right)\right), \quad \tilde{a}' \sim \pi_\theta\left(\cdot \mid s'\right)$$

12: Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y\left(r, s', d\right)\right)^2 \quad \text{for } i = 1, 2$$

13: Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}\left(s, \tilde{a}_\theta(s)\right) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s) \mid s\right)\right)$$

where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot \mid s)$ which is differentiable wrt θ via the reparametrization trick.

14: Update target network with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1-\rho)\phi_i \quad \text{for } i = 1, 2$$

15: **end for**
16: **end if**
17: **until** convergence or maximum training step reached

3.5 Twin Delayed Deep Deterministic Policy Gradient (TD3)

3.5.1 Several Improvements of TD3 based on DDPG

Although DDPG performs well in some tasks, the shortcomings of DDPG are also undeniable. One of the most common problems of DDPG is that it overestimates the Q-value when the critic network is updated, which misleads the actor network and makes the agent think that the current action is good enough and stop exploring for better actions.

In the actor-critic architecture, the role of the critic network is to learn the Q-value of actions, that is, to estimate the expected reward of the agent after making an action. Because the agent learning process is unsupervised, the Q-value is not limited, which leads to an abnormal "tacit agreement" between the actor network and the critic network, making the Q-value larger and larger, eventually resulting in a high but inaccurate Q-value.

On the other hand, the update of the critic network is a bootstrapping process; therefore, the Q-value will have bias, and the bias will get larger and larger as the bootstrapping begins to accumulate. The increasing bias also causes the Q-value learned by the agent to be higher than the actual Q-value.

To address the problem that DDPG often overestimates Q-values, Fujimoto et al. proposed TD3 [7] , which makes several improvements to the original DDPG.

3.5.1.1 Introduce twin Q-function

The DDPG algorithm learns only one Q-function. A single Q-function means it will be the only reference for the agent to update its policy. The Q-value of the single Q-function will be unconditionally referenced by the agent even if there is an overestimation problem.

In order to improve the reliability of Q-values referred by the agent, TD3 will learn two Q-functions simultaneously. The agent will refer to both Q-functions during the learning process and take the one with the smaller Q-value as the reference basis for policy updates. This approach addresses the problem of high Q-value. It weakens the agent's utilization of an action, potentially suggesting that the agent's current action may not be that good and motivating the agent to do more exploration.

3.5.1.2 Addressing Variance

As mentioned earlier, there is an error between the predicted Q-value and the true Q-value. Let the error between the predicted Q-value and the true Q-value be δ_t, then the expression of Q-value is:

$$\begin{aligned} Q_\theta\left(s_t, a_t\right) &= r_t + \gamma \mathbb{E}\left[Q_\theta\left(s_{t+1}, a_{t+1}\right)\right] - \delta_t \\ &= r_t + \gamma \mathbb{E}\left[r_{t+1} + \gamma \mathbb{E}\left[Q_\theta\left(s_{t+2}, a_{t+2}\right) - \delta_{t+1}\right]\right] - \delta_t \\ &= \mathbb{E}_{s_i \sim p_\pi, a_i \sim \pi}\left[\sum_{i=t}^{T} \gamma^{i-t}\left(r_i - \delta_i\right)\right] \end{aligned} \tag{3.18}$$

It can be seen that as the Q-function is continuously updated, its error will be accumulated. If the value of discount factor γ is large, then γ will further amplify the effect of overestimation and make the error larger. To address this problem, TD3 proposes several corresponding improvements.

Firstly, TD3 proposes a delayed-updated policy. TD3's idea is that since Q-functions have the problem of overestimation, updating policies in the actor network directly after each Q-function update will make policy updates more and more aggressive and eventually lead to more significant errors. However, if the Q-function can firstly update itself several times, then uses the relatively authentic Q-value to guide the policy update, the corresponding policy will be more reliable. The formula for updating the policy network is:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \tag{3.19}$$

where τ represents how often the policy is updated. When τ is 1, policy updates will be made every time the Q-function is updated. When τ becomes smaller and smaller, the frequency of policy updates will be lower and lower. The experiments in [7] prove that reducing the policy's update frequency can effectively reduce the overestimation.

3.5.1.3 Target Policy Smoothing Regularization

The authors of TD3 argue that similar actions should get similar reward values under the same state in a task. However, in the actual learning process of the agent, the reward values of similar actions can vary greatly. For this reason, an action noise mechanism is designed in TD3, i.e., after selecting the best action for the current state, a random Gaussian noise within a specified range is added to the action. The added Gaussian noise is:

$$\epsilon \sim \operatorname{clip}(\mathcal{N}(0, \sigma), -c, c) \tag{3.20}$$

where ϵ is a natural number in the range of $(-c, c)$.

By adding noise, each time the action used to calculate the Q-value will no longer be the best action, but the action is very close to the best action. The purpose of adding noise is to dodge the corresponding very high Q-value of an "excellent" action. Although the action has a very high Q-value, it is possible that similar actions do not have such a high Q-value, which implies that perhaps the Q-value for the "excellent" action should not be that high. So, by adding noise, it is possible to avoid the appearance of extreme Q-values by having the Q-value of the action taken to the surrounding area. The new Q-value then becomes:

$$y = r + \gamma Q_{\theta'}\left(s', \pi_{\phi'}\left(s'\right) + \epsilon\right) \tag{3.21}$$

3.5.2 TD3 implementation

First, TD3 maintains two critical networks and an actor network, then builds the target network correspondingly. Then, an action is generated, where the action is based on the current policy with random Gaussian noise, denoted as:

$$a \sim \pi_{\phi}(s) + \epsilon \tag{3.22}$$

Subsequently, the transition tuple after this action is generated, including the current state, the current action, the reward value and the next state, denoted as (s, a, r, s'). Then, the transition tuple is sampled from the replay buffer, the action in it will be added with Gaussian noise, and sent to each of the two Q target networks to derive the Q-value output by this action. Then, the smaller one of the two Q-values will be chosen to update the critic network. Finally, based on the update frequency τ, the critic network will update the policies in the actor network with a relatively lower frequency. A detailed procedure of TD3 is shown in Algorithm 5, where line 11 computes the target actions using the Target Policy Smoothing Regularization, line 12 computes the target Q-function, line 13 updates the Q-function by minimizing the MSBE, line 14 applies the delayed updating mechanism, and the policy will be updated in line 15.

3.6 Proximal Policy Optimization (PPO)

In 1992, Williams proposed the REINFORCE algorithm [29], using gradient estimates to update the policy by neural network-based RL. After that, Sutton et al. built a unifying framework that casts the previous algorithms as instances of the policy gradient method [19].

In continuous action domains, the agent cannot perform every possible action for learning because of the infinite number of available actions. In most cases, the agent can only rely on randomly selected actions for verification. In order to keep the policy update in a controllable range, Schulman et al. proposed Trust

Algorithm 5 Twin Delayed DDPG (TD3)

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$

1: Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1, \phi_{\text{targ},2} \leftarrow \phi_2$
2: **repeat**
3: Observe state s and select action $a = \text{clip}\left(\mu_\theta(s) + \epsilon, a_{\text{Low}}, a_{\text{High}}\right)$, where $\epsilon \sim \mathcal{N}$
4: Execute a in the environment
5: Observe next state s', reward r, and done signal d to indicate whether s' is terminal
6: Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
7: If s' is terminal, reset environment state.
8: **if** it's time to update **then**
9: **for** j in range (however many updates) **do**
10: Randomly sample a batch of transitions from $\mathcal{D}$
11: Compute target actions

$$a'\left(s'\right) = \text{clip}\left(\mu_{\theta_{\text{targ}}}\left(s'\right) + \text{clip}(\epsilon, -c, c), a_{\text{Low}}, a_{High}\right), \quad \epsilon \sim \mathcal{N}(0, \sigma)$$

12: Compute targets

$$y\left(r, s', d\right) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}\left(s', a'\left(s'\right)\right)$$

13: Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y\left(r, s', d\right)\right)^2 \quad \text{for } i = 1, 2$$

14: **if** j mod $policy - delay = 0$ **then**
15: Update policy by one step of gradient ascent using

$$\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1 - \rho)\phi_i \quad \text{for } i = 1, 2$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

16: **end if**
17: **end for**
18: **end if**
19: **until** convergence or maximum training step reached

Region Policy Optimization (TRPO) [17], which restricts each update to the policy in a trusted domain, thus significantly enhancing the stability of learning. Later, Schulman et al. proposed Proximal Policy Optimization (PPO) [18], which simplifies the trusted domain calculation process for TRPO. PPO and TRPO are two of the most prominent policy gradient algorithms used in deep RL, and the corresponding experiments confirm their versatility and robustness.

Unlike off-policy algorithms such as DDPG, SAC and TD3, PPO is an on-policy algorithm. The most significant difference between PPO and off-policy algorithms is how it collects data. Specifically, there is no replay buffer in PPO, which means that it does not save the experience learned before; PPO only learns the samples generated by the current policy and completely ignores the previous policy after generating a new policy. Meanwhile, PPO will conduct multiple rounds of learning on the samples generated by the current policy to fully tap the samples' learning potential, improve the samples' utilization, and constrain the difference between the old and new policies to ensure their steady improvement.

Usually, policy networks can be learned by policy gradient algorithms, such as Vanilla Policy Gradient [19]. Although the policy gradient algorithms are fast in learning, their performance is unstable. On the one hand, the policy gradient methods are sensitive to the setting of hyperparameters, and the setting of the learning rate has a significant impact on the results. On the other hand, the policy gradient algorithms will choose the policies more randomly, and the performance can fluctuate during the training process. The learning curves of the policy gradient algorithms usually contain a considerable amount of noise, and the whole curve fluctuates up and down sharply. In comparison, PPO is more stable than the traditional policy gradient algorithms. PPO uses the mechanism of trust region, which can limit the value of the final objective function in a controllable range even if the hyperparameters fluctuate greatly; therefore, PPO is less sensitive to hyperparameters than the other policy gradient algorithms, and the learning curve of PPO does not fluctuate drastically, which shows PPO is more robust than the others. Moreover, PPO is more sample efficient than traditional policy gradient algorithms; PPO can train better-performing policy networks when the same number of experiences are obtained.

3.6.1 Stochastic Gradient Ascent

In PPO, the goal is to find the optimal policy $\theta^\star$ that the agent can execute given the current environment and task, which is represented as:

$$\theta^\star = \underset{\theta}{\mathrm{argmax}} J(\theta) \tag{3.23}$$

where $J(l\theta)$ is the objective function, and θ is the current policy to be optimized.

In order to find the optimal policy $\theta^\star$, two steps need to be performed in Gradient Ascent. First, the

gradient of the objective function J under the current policy is obtained. The value of θ can be made larger by going in the objective direction. The expression for the gradient g is:

$$\mathbf{g} = \left.\frac{\partial J(\theta)}{\partial \theta}\right|_{\theta=\theta_{\text{old}}} \tag{3.24}$$

where θ_{old} represents the old policy learned during the last policy update.

After obtaining g, the old policy and the gradient can be used to learn the new policy. The new policy is denoted as:

$$\theta_{\text{new}} \leftarrow \theta_{\text{old}} + \alpha \cdot \mathbf{g} \tag{3.25}$$

where α is the learning rate. After repeating the above steps, a converged policy can be eventually obtained.

The premise of using Gradient Ascent is to find the gradient of the objective function J with respect to the policy θ. However, in some cases, the gradient cannot be found when J cannot be integrated, such as:

$$J(\theta) = \mathbb{E}_S[V(S;\theta)] \tag{3.26}$$

where S is the set of all states in the environment, V is the value function which judges the goodness of the current policy θ for the current task; $J(\theta)$ is the expectation of $V(S;\theta)$, implying that a definite integral is to be done for $V(S;\theta)$. But, the equation cannot be found as a definite integral, so the expectation and the gradient cannot be found.

However, the stochastic gradient can still be found. The stochastic gradient is a Monte Carlo approximation [13] to the expected value. In stochastic gradient ascent, a random sampling of the set of states S will be performed to derive its subset S'. The gradient is then computed using S', denoted as:

$$\mathbf{g} = \left.\frac{\partial V(S';\theta)}{\partial \theta}\right|_{\theta=\theta_{\text{old}}} \tag{3.27}$$

where g is the stochastic gradient, which represents the Monte Carlo approximation to the objective function. Finally, the new policy θ_{new} can be obtained as:

$$\theta_{\text{new}} \leftarrow \theta_{\text{old}} + \alpha \cdot \mathbf{g} \tag{3.28}$$

where α represents the discounted factor.

3.6.2 Trust Region

In the process of policy updating, because many parameters are involved, a slight change in one parameter can significantly affect the performance of the policy. Without limiting the magnitude of parameter updates, the performance of the policy will frequently fluctuate as the learning progresses, resulting in a decrease in the learning efficiency of the agent, such as the Vanilla Policy Gradient algorithm [19]. In order to solve the problem that the learning efficiency is affected by the fluctuation of the parameters, Schulman et al. [2015] proposed the concept of trust region [17], which sets a limit on the parameters involved in learning. The trust region limits the parameters involved in learning. Once the change in the parameters reaches the threshold value specified by the trust region, the parameters will stay at the specified threshold value. The trust region keeps the update of the new policy is within a controlled range and ensures the new policy does not differ too much from the old policy, thus achieving more stable training.

The optimization object of the trust region algorithm is $J(\theta)$, which is the same as the policy gradient algorithms. Now, $\mathcal{N}\left(\theta_{\text{old}}\right)$ is used to denote the neighborhood of the current policy, which represents the set of all similar policies in the neighborhood of the current policy. The expression of $\mathcal{N}\left(\theta_{\text{old}}\right)$ is:

$$\mathcal{N}\left(\theta_{\text{old}}\right)=\left\{\theta \mid \left\|\theta-\theta_{\text{old}}\right\|_2 \leq \Delta\right\} \tag{3.29}$$

where Δ is a positive real number and the inequality in curly brackets represents that the distance between the current policy θ and the old policy θ_{old} cannot exceed Δ. In other words, if the old policy θ_{old} is the center point of a circle, Δ represents the radius of the circle, then the location of the new policy θ will always stay in this circle.

To summarize, the meaning of trust region is that suppose there is another objective function $L\left(\theta \mid \theta_{\text{old}}\right)$ which is located very close to the original objective function $J(\theta)$ in the neighborhood of θ_{old}, then the neighborhood $\mathcal{N}\left(\boldsymbol{\theta}_{\text{old}}\right)$ of θ_{old} is called the trust region, and the new policy will be accepted at any point within the range of the trust region.

In general, the form of $J(\theta)$ is complex, and it is not conducive to computation. However, in the trust region, the form of $L\left(\theta \mid \theta_{\text{old}}\right)$ is more concise than $J(\theta)$, so the computation can start by letting $L\left(\theta \mid \theta_{\text{old}}\right)$ be close to $J(\boldsymbol{\theta})$ in the range of the trust region, and then use the function $L\left(\theta \mid \theta_{\text{old}}\right)$ instead of $J(\theta)$ to reduce the computational complexity of policy optimization.

The algorithms in trust region are performed in two steps. The first step is approximation, given the old policy θ_{old} and constructing the objective function $L\left(\theta \mid \boldsymbol{\theta}_{\text{old}}\right)$. The method of constructing $L\left(\theta \mid \theta_{\text{old}}\right)$ can use either the second-order Taylor's expansion [4] of $J(\theta)$ or the Monte Carlo approximation [13] of $J(\theta)$. The second step is maximization, where the maximum value of $L\left(\boldsymbol{\theta} \mid \boldsymbol{\theta}_{\text{old}}\right)$ is searched in the trust region

and assigned to the new policy, denoted as:

$$\theta_{\text{new}} \leftarrow \underset{\theta \in \mathcal{N}(\theta_{\text{old}})}{\operatorname{argmax}} L\left(\theta \mid \theta_{\text{old}}\right) \tag{3.30}$$

3.6.3 PPO Implementation

The working process of PPO is similar to the policy gradient algorithms. In order to solve the problem of unstable performance of the traditional policy gradient algorithms and reduce the impact of hyperparameters, PPO introduces the trust region mechanism, which controls the difference between the new policy and the old policy within a range and improves the overall learning stability.

There are two forms of PPO to limit the magnitude of policy updates, one is PPO with KL-divergence in the objective function [12], and the other is PPO with clip in the objective function [18]. The clip-based PPO is applied in the experiments of this thesis.

First, the policy in PPO is updated in the same way as the policy gradient algorithm, which is to maximize the objective function of the policy, denoted as:

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a \sim \pi_{\theta_k}}{\mathrm{E}} \left[L\left(s, a, \theta_k, \theta\right)\right] \tag{3.31}$$

where θ_{k+1} is the new policy, and $L\left(s, a, \theta_k, \theta\right)$ is the approximation function of the actual objective function $J(\boldsymbol{\theta})$ in the trust region. Because of the inclusion of the clip mechanism in PPO, the new $L\left(s, a, \theta_k, \theta\right)$ will be expressed as:

$$L\left(s, a, \theta_k, \theta\right) = \min\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s, a), \operatorname{clip}\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s, a)\right) \tag{3.32}$$

where ϵ is a hyperparameter used to clip so as to limit the distance between the new policy and the old one. $A^{\pi_{\theta_k}}(s, a)$ is called advantage function and is used to evaluate how good the action a is in the current state s throughout the trajectory of the current policy.

For a more intuitive understanding of clip, $L\left(s, a, \theta_k, \theta\right)$ has a more concise expression:

$$L\left(s, a, \theta_k, \theta\right) = \min\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s, a), \quad g\left(\epsilon, A^{\pi_{\theta_k}}(s, a)\right)\right) \tag{3.33}$$

where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases}$$

It can be seen that during the evaluation of $L(s, a, \theta_k, \theta)$, if the value of the advantage function $A^{\pi_{\theta_k}}(s, a)$ is positive, it means that the action a can bring positive benefits in the current state. Then the algorithm will default that the appearance frequency of a in s will increase in the subsequent training process, and consequently, the value of the objective function also increases. At this time, the expression of the approximated objective function is:

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)}, (1+\epsilon)\right) A^{\pi_{\theta_k}}(s, a) \tag{3.34}$$

However, because PPO does not want the value of the objective function to change too drastically and thus potentially mislead the agent, it will limit the growth of the objective function. Once $\pi_\theta(a \mid s) > (1+\epsilon)\pi_{\theta_k}(a \mid s)$, the value of the objective function will reach a ceiling, and the objective function will take the value no more than $(1+\epsilon)A^{\pi_{\theta_k}}(s, a)$. In this way, limiting the update magnitude of the objective function also limits the distance of the new policy from the old one.

In the second case, if the value of the advantage function $A^{\pi_{\theta_k}}(s, a)$ is negative, this means that the action in the current state is not ideal and does not bring positive benefits to the whole policy. At this point, the algorithm will default to the fact that the frequency of the action a will decrease in the subsequent training process in state s; hence, the value of the objective function will increase. The expression of the objective function will become:

$$L(s, a, \theta_k, \theta) = \max\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)}, (1-\epsilon)\right) A^{\pi_{\theta_k}}(s, a) \tag{3.35}$$

Similar to when the advantage function is positive, when $\pi_\theta(a \mid s) < (1-\epsilon)\pi_{\theta_k}(a \mid s)$, the value of the objective function will reach a ceiling and the objective function will take no more than $(1-\epsilon)A^{\pi_{\theta_k}(s,a)}$, thus also limiting the update magnitude of the policy. The peudocode of PPO is shown in Algorithm 6, where line 5 updates the policy by adding the trust region to ensure that the new policy does not benefit by going far away from the old policy, and then it will update the corresponding value function of the current policy in line 6.

Algorithm 6 Proximal Policy Optimization (PPO)

Input: initial policy parameters θ_0, initial value function parameters ϕ_0

1: **for** $k = 0, 1, 2, \ldots$ **do**

2: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

3: Compute the reward $\hat{R}_t$

4: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

5: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_k}(a_t \mid s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right)$$

via stochastic gradient ascent with Adam optimizer.

6: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2$$

via gradient descent algorithm.

7: **end for**

Chapter 4

New Algorithms for Reward Machines in Deep Reinforcement Learning

Currently, the deep RL algorithms combined with reward machines in related work are DDPG and the option-based Hierarchical Reinforcement Learning (HRL). However, with the constant development of deep RL algorithms, the performance of DDPG and HRL has been surpassed by newer algorithms in many testing environments. Therefore, in order to further improve the learning efficiency of RM-based algorithms, we choose three mainstream deep RL algorithms: Soft Actor-Critic (SAC) [10], Twin Delayed Deep Deterministic Policy Gradient (TD3) [7], and Proximal Policy Optimization (PPO) [18] as our baseline algorithms, combine them with CRM and evaluate their performance. The new algorithms are called SAC-CRM, TD3-CRM, and PPO-CRM, respectively[1].

4.1 Soft Actor-Critic (SAC) with CRM

First, we use SAC as the baseline algorithm and propose a new algorithm, SAC-CRM, that exploits the task structure made visible by the RM. In SAC-CRM, the agent still uses the entropy value from the baseline SAC when updating the Q-function and continues the Energy Based Policy model from the baseline SAC. In contrast to the baseline, SAC-CRM changes the type of learning experience compared to the baseline SAC and adds CRM's counterfactual experience. The pseudocode of SAC-CRM is shown in Algorithm 7.

In combination with reward machines, the learning environment of SAC-CRM becomes an MDPRM, so the RM experience in the RM environment will be added to the experience. The actual experience learned

[1] The code of SAC-CRM, TD3-CRM, and PPO-CRM, as well as all the tasks mentioned in this thesis, are available at `https://github.com/haolinsun0907/Exploiting_Reward_Machines_with_Deep_Reinforcement_Learning`

by the agent will change from the original $\langle \mathrm{s, a, r, s'} \rangle$ to $\langle \mathrm{s}, \bar{u}, \mathrm{a}, \bar{r}, \mathrm{s'}, \bar{u}' \rangle$, where $\bar{u}$ and $\bar{u}'$ are the RM states before and after the action a. Also, CRM will generate a set of counterfactual experiences for each RM state after the agent takes an action, which is shown in line 7 of the pseudocode.

To generate the counterfactual experience, the agent will traverse each RM state $\bar{u} \in U$ after making an action. If the agent's action in $\bar{u}$ causes the environmental state s change to the next environmental state s', then the next RM state will become $\bar{u}' = \delta_u\left(\bar{u}, L\left(s, a, s'\right)\right)$, and the agent will receive a reward given by the RM, which is $\bar{r} = \delta_r(\bar{u})\left(s, a, s'\right)$. The CRM generates a set of counterfactual experiences whose expression is:

$$\left\{\left(s, \bar{u}, a, \delta_r(\bar{u})\left(s, a, s'\right), s', \delta_u\left(\bar{u}, L\left(s, a, s'\right)\right)\right) \mid \bar{u} \in U\right\}$$

Correspondingly, SAC-CRM will learn the information provided by CRM when updating the policy. Specifically, the agent will consider both the actual experience and counterfactual experiences. In terms of reward, the agent will now consider the RM reward provided by the state-reward function. At this point, since the agent learns in an MDPRM, SAC-CRM will not only consider the actual environmental state but the cross-product of the environmental state and the RM state, as well as the counterfactual experiences provided by CRM (line 9 to line 15 in the pseudocode). As such, the soft Q-function used to evaluate the policies in SAC-CRM is changed from equation 3.11 to:

$$Q^{\pi}_{soft}(s, \bar{u}, a) = \underset{s_t, \bar{u}_t, a_t \sim \rho_\pi}{\mathbb{E}}\left[\sum_{t=0}^{\infty} \gamma^t \bar{r}\left(s_t, \bar{u}_t, a_t\right) + \alpha \sum_{t=1}^{\infty} \gamma^t H\left(\pi\left(\cdot \mid s_t, \bar{u}_t\right)\right) \mid s_0 = s, \bar{u}_0 = u, a_0 = a\right] \tag{4.1}$$

which considers the cross product of the environmental state s_t and RM state $\bar{u}_t$ at time t. Similarly, the Energy Based Policy model also changes from equation 3.14 to:

$$\pi\left(a_t \mid s_t, \bar{u}_t\right) \propto \exp\left(\frac{1}{\alpha} Q_{soft}\left(s_t, \bar{u}_t, a_t\right)\right) \tag{4.2}$$

Finally, SAC-CRM will aim to minimize the value of the loss function based on both the actual experience and the counterfactual experience (line 15 in the pseudocode). Thus, the new loss function for the temporary policy π_ϕ in the RM setting can be obtained from equation 3.17 as follows:

$$\begin{aligned} J_\pi(\phi) &= D_{\mathrm{KL}}\left(\pi_\phi\left(\cdot \mid s_t, \bar{u}_t\right) \| \exp\left(\frac{1}{\alpha} Q_\theta\left(s_t, \bar{u}_t, \cdot\right) - \log Z\left(s_t, \bar{u}_t\right)\right)\right) \\ &= \mathbb{E}_{s_t, \bar{u}_t \sim \mathcal{D}, a_t \sim \pi_\phi}\left[\log\left(\frac{\pi_\phi\left(a_t \mid s_t, \bar{u}_t\right)}{\exp\left(\frac{1}{\alpha} Q_\theta\left(s_t, \bar{u}_t, a_t\right) - \log Z\left(s_t, \bar{u}_t\right)\right)}\right)\right] \\ &= \mathbb{E}_{s_t, \bar{u}_t, \sim \mathcal{D}, a_t \sim \pi_\phi}\left[\log \pi_\phi\left(a_t \mid s_t, \bar{u}_t\right) - \frac{1}{\alpha} Q_\theta\left(s_t, \bar{u}_t, a_t\right) + \log Z\left(s_t, \bar{u}_t\right)\right] \end{aligned} \tag{4.3}$$

Algorithm 7 Soft Actor-Critic with counterfactual experiences for RMs (CRM).

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$, labelling function L, a finite set of states U, a finite set of terminal states F, state-transition function δ_u, state-reward function δ_r, initial RM state $u_0 \in U$

1: Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1, \phi_{\text{targ},2} \leftarrow \phi_2$
2: Initialize $u \leftarrow u_0$ and $s \leftarrow$ EnvInitialState()
3: **repeat**
4: Observe state s and select action $a \sim \pi_\theta(\cdot \mid s, u)$
5: Execute a in the environment and observe next state s'
6: Compute the reward $r \leftarrow \delta_r(u)\,(s, a, s')$ and next RM state $u' \leftarrow \delta_u\,(u, L\,(s, a, s'))$, and done signal d to indicate whether s' is terminal
7: Set experience $\leftarrow \{(s, \bar{u}, a, \delta_r(\bar{u})\,(s, a, s')\,, s', \delta_u\,(\bar{u}, L\,(s, a, s'))\,, d) \mid \bar{u} \in U\}$
8: Store experience in replay buffer $\mathcal{D}$
9: If s' is terminal or $\bar{u} \in F$, reset environment state.
10: **if** it's time to update **then**
11: **for** j in range (however many updates) **do**
12: Randomly sample a batch B of transitions from $\mathcal{D}$
13: Compute targets for the Q functions:

$$y\,(\bar{r}, s', \bar{u}', d) = r + \gamma(1-d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}\,(s', \bar{u}', \tilde{a}') - \alpha \log \pi_\theta\,(\tilde{a}' \mid s', \bar{u}')\right), \quad \tilde{a}' \sim \pi_\theta\,(\cdot \mid s', \bar{u}')$$

14: Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,\bar{u},a,\bar{r},s',\bar{u}',d) \in B} \left(Q_{\phi_i}(s, \bar{u}, a) - y\,(\bar{r}, s', \bar{u}', d)\right)^2 \qquad \text{for } i = 1, 2$$

15: Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s,\bar{u} \in B} \left(\min_{i=1,2} Q_{\phi_i}\,(s, \bar{u}, \tilde{a}_\theta(s, \bar{u})) - \alpha \log \pi_\theta\,(\tilde{a}_\theta(s, \bar{u}) \mid s, \bar{u})\right)$$

where $\tilde{a}_\theta(s, \bar{u})$ is a sample from $\pi_\theta(\cdot \mid s, \bar{u})$ which is differentiable wrt θ via the reparametrization trick.
16: Update target network with

$$\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1-\rho)\phi_i \qquad \text{for } i = 1, 2$$

17: **end for**
18: **end if**
19: **until** convergence or maximum training step reached

4.2 Twin Delayed Deep Deterministic Policy Gradient (TD3) with CRM

Then, the next baseline algorithm we choose to adapt with CRM is Twin Delayed Deep Deterministic Policy Gradient (TD3). The pseudocode of TD3-CRM is shown in Algorithm 8. Because the TD3 algorithm has a similar update process in the Q-function and policy network as DDPG, combining TD3 with CRM is similar to DDPG. First, because the learning environment becomes an MDPRM, the information from the reward machine needs to be added to the learning experience. That is, we use the cross-product states to replace the environmental states in the actual experience. As such, the agent's actual experience will be changed from $\langle s, a, r, s' \rangle$ to $\langle s, u, a, r, s', u' \rangle$. Also, because CRM has been added, CRM will generate a corresponding counterfactual experience for each RM state after the agent executes each action (line 7 of the pseudocode) which contains the next RM state $\bar{u}'$ calculated using the state-transition function $\delta_u(\bar{u})$, and the RM reward $\bar{r}$ calculated by the state-reward function $\delta_r(\bar{u})$. The set of counterfactual experiences is expressed as:

$$\{(s, \bar{u}, a, \delta_r(\bar{u})(s, a, s'), s', \delta_u(\bar{u}, L(s, a, s'))) \mid \bar{u} \in U\}$$

Following the updated experiences, the actor and critic networks of TD3-CRM will be updated with reference to the current environmental state s and the next environmental state s', as well as the current RM state $\bar{u}$ and the next RM state $\bar{u}'$ derived from $\delta_u(\bar{u})$. In terms of reward, TD3-CRM will refer to the RM reward $\bar{r}$ provided by the state-reward function $\delta_r(\bar{u})$. Specifically, both of the Q-functions of TD3-CRM are changed from equation 3.18 to:

$$\begin{aligned} Q_\theta(s_t, \bar{u}_t, a_t) &= \bar{r}_t + \gamma \mathbb{E}[Q_\theta(s_{t+1}, \bar{u}_{t+1}, a_{t+1})] - \delta_t \\ &= \bar{r}_t + \gamma \mathbb{E}[\bar{r}_{t+1} + \gamma \mathbb{E}[Q_\theta(s_{t+2}, \bar{u}_{t+2}, a_{t+2}) - \delta_{t+1}]] - \delta_t \\ &= \mathbb{E}_{s_i, \bar{u}_i, \sim p_\pi, a_i \sim \pi}\left[\sum_{i=t}^{T} \gamma^{i-t}(\bar{r}_i - \delta_i)\right] \end{aligned} \tag{4.4}$$

Here, the RM state $\bar{u}$ is added for the update reference, and the reward is changed from the original environmental reward r to the RM reward $\bar{r}$.

Accordingly, the Q-value with the addition of Gaussian noise will also refer to the next RM state as well as the RM reward, changing from equation 3.21 to

$$y = \bar{r} + \gamma Q_{\theta'}(s', \bar{u}', \pi_{\phi'}(s', \bar{u}') + \epsilon) \tag{4.5}$$

As we can see in the pseudocode of TD3-CRM, in line 6, the corresponding RM state and the RM reward are calculated after the agent has taken an action. In lines 7 and 8, the counterfactual experience generated by CRM is added to the agent's replay buffer. From line 10 to line 17, TD3-CRM updates its policies in the same way as the original TD3, where line 13 computes the target actions using the Target Policy Smoothing Regularization, line 14 computes the target Q-function, line 15 updates the Q-function by minimizing the MSBE, line 16 applies the delayed updating mechanism, and the policy will be updated in line 17 with the reference of both the actual experience and the counterfactual experience.

Algorithm 8 TD3 with counterfactual experiences for RMs (CRM).

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$, labelling function L, a finite set of states U, a finite set of terminal states F, state-transition function δ_u, state-reward function δ_r, initial RM state $u_0 \in U$

1: Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1, \phi_{\text{targ},2} \leftarrow \phi_2$
2: Initialize $u \leftarrow u_0$ and $s \leftarrow$ EnvInitialState()
3: **repeat**
4: Observe state s and select action $a = \text{clip}\left(\mu_\theta(s) + \epsilon, a_{\text{Low}}, a_{\text{High}}\right)$, where $\epsilon \sim \mathcal{N}$
5: Execute a in the environment and observe next state s'
6: Compute the reward $r \leftarrow \delta_r(u)\left(s, a, s'\right)$ and next RM state $u' \leftarrow \delta_u\left(u, L\left(s, a, s'\right)\right)$, and done signal d to indicate whether s' is terminal
7: Set experience $\leftarrow \left\{\left(s, \bar{u}, a, \delta_r(\bar{u})\left(s, a, s'\right), s', \delta_u\left(\bar{u}, L\left(s, a, s'\right)\right), d\right) \mid \bar{u} \in U\right\}$
8: Store experience in replay buffer $\mathcal{D}$
9: If s' is terminal or $\bar{u} \in F$, reset environment state.
10: **if** it's time to update **then**
11: **for** j in range (however many updates) **do**
12: Randomly sample a batch B of transitions from $\mathcal{D}$
13: Compute target actions

$$a'\left(s', \bar{u}'\right) = \text{clip}\left(\mu_{\theta_{\text{targ}}}\left(s', \bar{u}'\right) + \text{clip}(\epsilon, -c, c), a_{\text{Low}}, a_{High}\right), \quad \epsilon \sim \mathcal{N}(0, \sigma)$$

14: Compute targets

$$y\left(\bar{r}, s', \bar{u}', d\right) = \bar{r} + \gamma(1-d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}\left(s', \bar{u}', a'\left(s', \bar{u}'\right)\right)$$

15: Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,\bar{u},a,\bar{r},s',\bar{u}',d) \in B} \left(Q_{\phi_i}(s, \bar{u}, a) - y\left(\bar{r}, s', \bar{u}', d\right)\right)^2 \quad \text{for } i = 1, 2$$

16: **if** j mod $policy - delay = 0$ **then**
17: Update policy by one step of gradient ascent using

$$\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1-\rho)\phi_i \quad \text{for } i = 1, 2$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$$

18: **end if**
19: **end for**
20: **end if**
21: **until** convergence or maximum training step reached

4.3 Proximal Policy Optimization (PPO) with CRM

The last baseline algorithm we choose to adapt with CRM is called Proximal Policy Optimization (PPO). Integrating CRM into the PPO algorithm is unique and different from SAC and TD3. First, because PPO is an on-policy algorithm, there is no replay buffer in PPO to store the experience previously learned by the agent. Unlike the off-policy algorithms, PPO is learned by generating multiple different trajectories after randomly sampling the state distribution and actions of the whole task. A trajectory represents a sequence of states and actions, including a possible sequence of states that the agent will experience during the execution of the task and the actions the agent will perform in each state. Specifically, a trajectory τ can be represented as:

$$\tau = (s_0, a_0, s_1, a_1, \ldots) \tag{4.6}$$

where the initial state s_0 is randomly sampled from the initial state distribution. Subsequently, the next state s_{t+1} can be derived from the action a_t selected by the current policy in the current state s_t. And so on, a complete trajectory performed by the agent in a task can be obtained.

The form of the trajectory indicates that in PPO, the agent learns via planned routes, implying that the single-step experience generation approach used in off-policy algorithms is no longer applicable in PPO. As a result, if the agent wishes to perform PPO in an MDPRM while using CRM, the RM states for each environmental state s must be computed using the state-transition function δ_u. As the learning proceeds, PPO must calculate the corresponding RM state and add it to the trajectories constantly. Furthermore, CRM will generate one counterfactual experience for each RM state after each action. However, unlike the off-policy algorithms, PPO does not have a replay buffer to store the counterfactual experiences. Thus, we made several changes to adapt CRM into PPO.

First, in order for the agent to learn in an MDPRM, we switch the learning environment from a traditional MDP to an MDPRM. As such, all the trajectories will include the environmental states and the corresponding RM states, and the reward will become the RM reward. Specifically, in an MDPRM, an actual trajectory τ is a sequence:

$$(s_0, u_0, a_0, s_1, u_1, a_1, \ldots, s_k, u_k, a_k, s_{k+1}, u_{k+1}) \tag{4.7}$$

such that

$$u_{i+1} = \delta_u(u_i, L(s_i, a_i, s_{i+1})), \forall i : 0 < i \leq k \tag{4.8}$$

The reward of such a trajectory is:

$$r(\tau) = \sum_{i=0}^{k} \gamma^i \delta_r(u_i)(s_i, a_i, s_{i+1}) \tag{4.9}$$

where γ is the discount factor.

Then, we adapt the counterfactual experiences into the form of the trajectories. As mentioned before, CRM will generate one counterfactual experience per RM state after each action. The form of the counterfactual experience is the same as in SAC-CRM and TD3-CRM, which is:

$$\{(s, \bar{u}, a, \delta_r(\bar{u})\,(s, a, s')\,, s', \delta_u\,(\bar{u}, L\,(s, a, s'))) \mid \bar{u} \in U\}$$

However, this experience cannot be used by PPO directly since it is a different form from the trajectory. Thus, we extract some useful information from the original counterfactual experiences and recombine them into the form of the trajectory.

For every trajectory sampled from the counterfactual experiences, we take the cross product of the current environmental state s and the current RM state $\bar{u}$ as the initial state and use the current policy to generate the rest of the trajectory. Since this form of trajectory is sampled from counterfactual experiences, we call it a *counterfactual trajectory*. As such, the set of counterfactual trajectories is:

$$C(\tau) = \{(s_0, u'_0, a_0, s_1, u'_1, a_1, \ldots, s_k, u'_k, a_k, s_{k+1}, u'_{k+1}) | u'_i \in U, \forall i : 0 < i \leq k, u'_{k+1} \in U \cup F\} \tag{4.10}$$

and the number of such counterfactual trajectories is $|u|^k$. Then, we add $C(\tau)$ to the original trajectory set and use both the actual trajectories and the counterfactual trajectories to do the policy updates. Since the number of counterfactual trajectories can quickly become very large, one could consider ways of constraining them.

Subsequently, during the policy update, the maximum value of the objective function will be determined by the environmental state s as well as the RM state $\bar{u}$ because of the addition of the RM information. The new objective function will change from equation 3.31 to:

$$\theta_{k+1} = \arg\max_{\theta} \mathop{\mathrm{E}}_{s, \bar{u}, a \sim \pi_{\theta_k}} \left[L\,(s, \bar{u}, a, \theta_k, \theta)\right] \tag{4.11}$$

Also, in the calculation of the approximation function, the policy π will consider the RM information, the

new approximation function is changed from equation 3.32 to:

$$L\left(s, \bar{u}, a, \theta_k, \theta\right) = \min\left(\frac{\pi_\theta(a \mid s, \bar{u})}{\pi_{\theta_k}(a \mid s, \bar{u})} A^{\pi_{\theta_k}}(s, \bar{u}, a), \operatorname{clip}\left(\frac{\pi_\theta(a \mid s, \bar{u})}{\pi_{\theta_k}(a \mid s, \bar{u})}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s, \bar{u}, a)\right) \tag{4.12}$$

The pseudocode of PPO-CRM is shown in Algorithm 9. In line 3, the set of actual trajectories is collected. In line 4, all the counterfactual experiences will be converted to counterfactual trajectories and be added to the actual trajectory set. From line 5 to line 8, PPO-CRM will update in the same way as the original PPO, where line 7 updates the policy by adding the trust region to ensure that the new policy does not benefit by going far away from the old policy, and then it will update the corresponding value function of the current policy in line 8 with the reference of both the actual trajectories and the counterfactual trajectories.

Algorithm 9 PPO with counterfactual experiences for RMs (CRM).

Input: initial policy parameters θ_0, initial value function parameters ϕ_0, labelling function L, a finite set of states U, a finite set of terminal states F, state-transition function δ_u, state-reward function δ_r, initial RM state $u_0 \in U$

1: Initialize $u \leftarrow u_0$ and $s \leftarrow$ EnvInitialState()
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of actual trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Collect set of counterfactual trajectories $C(\tau)$ and add $C(\tau)$ to $\mathcal{D}_k$.
5: Compute the RM reward $\hat{R}_t$.
6: Compute advantage estimates $\hat{A}_t$ based on the current value function V_{ϕ_k}.
7: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta\left(a_t \mid s_t, \bar{u}_t\right)}{\pi_{\theta_k}\left(a_t \mid s_t, \bar{u}_t\right)} A^{\pi_{\theta_k}}\left(s_t, \bar{u}_t, a_t\right), \quad g\left(\epsilon, A^{\pi_{\theta_k}}\left(s_t, \bar{u}_t, a_t\right)\right)\right)$$

via stochastic gradient ascent with Adam optimizer.

8: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi\left(s_t, \bar{u}_t\right) - \hat{R}_t\right)^2$$

via gradient descent algorithm.

9: **end for**

4.4 Discussion

Overall, adapting CRM with off-policy deep RL algorithms such as SAC and TD3 is relatively straightforward. There are two major things we need to do to combine CRM with such off-policy algorithms. First, we need to change the learning environment from a traditional MDP to an MDPRM. Then, we need to add the counterfactual experiences to the replay buffer in addition to the actual experience. The main challenge we faced when adapting CRM with off-policy algorithms is to understand the update mechanism of each

algorithm and think about whether they are going to have any synergy with CRM.

On the other hand, adapting CRM with an on-policy deep RL algorithm is not trivial. An on-policy algorithm such as PPO does not have a replay buffer, and since there is no place to store the previously learned experience, counterfactual experience cannot be utilized properly. Moreover, on-policy algorithms usually learn through trajectories, which have low sampling efficiency, and the generation process of counterfactual experience requires a lot of data input and output, which will further increase the sampling time. Last but not least, since the expression form of trajectories is different from that of counterfactual experience, we also need to adapt the form of counterfactual experience to the form of the trajectories to make the learning proceed, which will lose some useful information, such as the single-step reward value. Based on the above, it seems that adding counterfactual experience to an on-policy algorithm can provide little help to the algorithm but further increase the sampling time.

Chapter 5

Experimental Evaluation

In this chapter, we will test the proposed new algorithms in continuous action domains and compare the performance of the new algorithms with the existing algorithms for reward machines. The new algorithms include SAC-CRM, TD3-CRM, and PPO-CRM, while the existing algorithms include DDPG-CRM and HRM [23]. The test environments are two different types of continuous action domains in OpenAI gym, which are the Half-Cheetah domain in 2D and the Ant domain in 3D [3]. All CRM-based algorithms have a batch size of $100n$, where $n = |U|$ and U is the number of non-terminal RM states. For HRM, the option policies are learned using DDPG, and the high-level policy uses DQN. In the following experiments, because HRM uses deep neural networks to learn the policies, we will use *Option-DHRM* as the legend name to represent the results. In Option-DHRM, the batch size is $100n$, where n represents the number of available options. The neural networks of all the algorithms in this thesis use two hidden layers, each with 256 units with *RELU* as the activation function. The CPU used for all experiments was the Intel Core i9-9900K with a maximum frequency of 5 GHz, the GPU was the NVIDIA GeForce RTX 2080 with 8 GB of video memory, and the memory was 32 GB of DDR4 at a maximum speed of 3000 MHz. Also, we expect each trial of all the algorithms to finish the training process within ten hours, so we stop each trial after ten hours.

In both domains, the agent's task is to reach a number of different points located in different locations in a specific order. We define this type of task for two purposes. One purpose is to examine the capability of each algorithm to control the movement of the agent through the cooperation between the limbs of the agent. Since both the cheetah robot in the Half-Cheetah domain and the ant robot in the Ant domain move the entire body through cooperation between its limbs, the degree of cooperation between the robots' limbs directly affects their movement speed, thus affecting the number of steps required for the robots to reach the specified point. In the body control part, the learning efficiency of the baseline algorithms will determine

how fast the robots can move after learning within the specified maximum number of learning steps. Another purpose is to examine how much CRM can help the agent complete the task. Because the environments in the experiment are RM environments, and a task contains multiple target points with high complexity, it would be challenging for the agent to complete the task only using the cross-product baseline of reward machines. However, because the addition of CRM can give the agent more specific information about the task, the efficiency of the agent in completing the task of reaching multiple target points will be improved with CRM.

5.1 Results in the Half-Cheetah Domain

The first domain in our experiments is Half-Cheetah, which has a continuous action space. In this environment, the agent is a cheetah-like robot. The robot has six joints that it must learn to control to stand up, move forward, or backwards. The robot chooses the moving angle and the amount of force to apply to each joint at each step, making the action space infinite. The state space is also continuous, including the location and velocities of each joint. An abstract representation of this domain is shown in Figure 5.1, which includes the starting position of the cheetah robot and the points distribution.

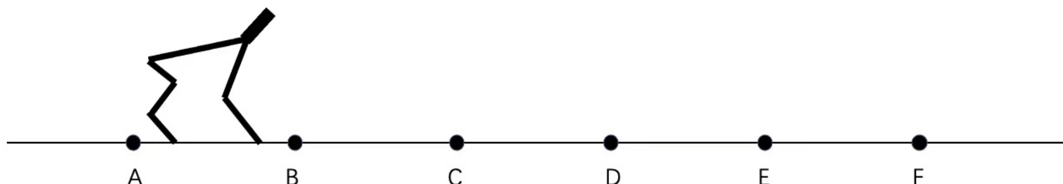

Figure 5.1: The Half-Cheetah domain

In this domain, we will be testing the new algorithms on five tasks including the original two tasks defined by Toro Icarte et al. [23]. Table 5.1 shows the tasks and their descriptions. We will use these tasks to test and compare the performance of the proposed new algorithms, which are SAC-CRM, TD3-CRM and PPO-CRM. We will also compare the performance of new algorithms with the existing algorithms for reward machines, which are DDPG-CRM and Option-DHRM. For all the tasks, the agent will start from an arbitrary position between point A and point B. At the same time, to give the agent more reference and prevent the agent from stopping to explore new actions, after each transition of the RM state, the agent will receive a small negative reward value, denoted as *Control Penalty (CP)*.

Task 1 is to have the agent go back and forth between points A and B as many times as possible in 1000 training steps. The task's RM automaton is shown in Figure 5.2.

Table 5.1: Tasks for the Half-CHeetah domain.

Task No.	Description
1	Start from an arbitrary position between points A and B, first go to point B, then repeatedly go back and forth between A and B.
2	Start from an arbitrary position between points A and B, first go to point A, then to point B then to point C, and then back to point B and then to point A and then repeat indefinitely.
3	Start from an arbitrary position between points A and B, first go to point A, then to point B then to point C, go back to point B, then return to point C, then reach point D and stop.
4	Start from an arbitrary position between points A and B, either go to point A or to point B, then go to point C and then reach point D and stop.
5	Start from an arbitrary position between points A and B, first go to point B, then pass through points C, D, and E, and then reach point F and stop.

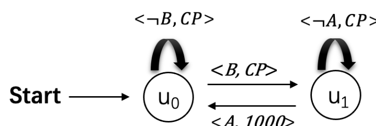

Figure 5.2: The RM automaton of Task 1 in the Half-Cheetah domain.

Task 2 is a more complex version of Task 1, which increases the distance between the leftmost point (point A) and the rightmost point (point C). In order to take advantage of the fact that reward machines can decompose tasks, we added an intermediate point between point A and point C. This allows the task of going back and forth between point A and point C to be decomposed into first going from point A to point B and then from point B to point C, which shortens the distance between the points and allows the agent to use less time to explore and thus learn to complete the task more efficiently. The RM automaton of the second task is shown in Figure 5.3.

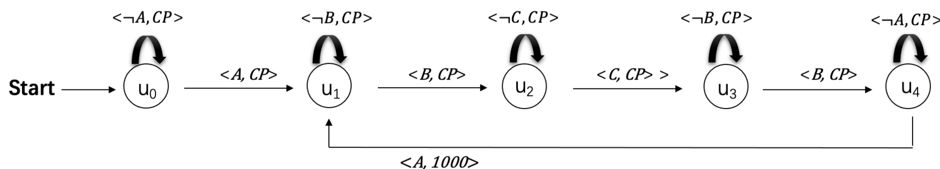

Figure 5.3: The RM automaton of Task 2 in the Half-Cheetah domain.

Task 3 further increases the distance between the starting point and the ending point and adds the operation of turning back halfway. The agent can make full use of the counterfactual experience of CRM in

this task; after finishing the task of turning back, the agent can continue forward more quickly through the experience already learned. The RM automaton of Task 3 is shown in Figure 5.4.

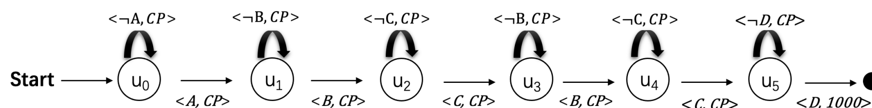

Figure 5.4: The RM automaton of Task 3 in the Half-Cheetah domain.

Task 4 adds alternatives to the target points of the agent's subtasks, demonstrating reward machines' ability to define complex tasks. The agent can choose among different subtasks by following the task description given by the RM and completing the whole task under the RM's guidance. The RM automaton of Task 4 is shown in Figure 5.5.

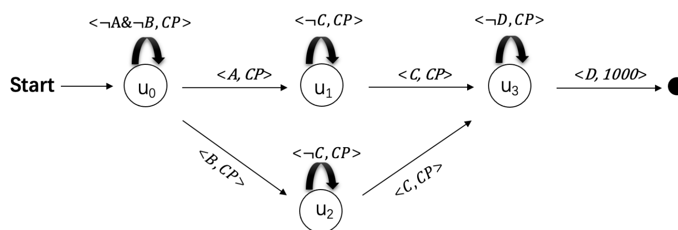

Figure 5.5: The RM automaton of Task 4 in the Half-Cheetah domain.

Task 5 is the most challenging task in the Half-Cheetah domain. The RM automaton of Task 5 shown in Figure 5.6. First, the distance between the starting and ending points is the longest of all tasks. Although

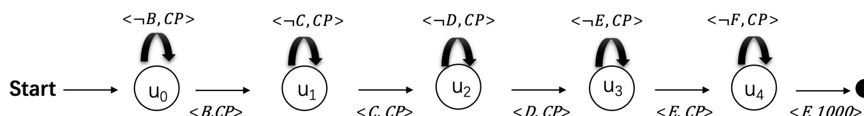

Figure 5.6: The RM automaton of Task 5 in the Half-Cheetah domain.

we can use RM to add multiple intermediate points between the starting and ending points to shorten the distance in each subtask, because of the considerable distance between the starting and ending points and sparser reward, the agent needs to do a lot of exploration. More importantly, in this task, the agent's goal is to keep moving forward without any backward step, so the counterfactual experience learned in CRM has little effect on this task. Task 5 is designed more to verify the suitability of CRM for this specific type of task.

During the training process, the algorithms' exploration of the actions is random, which leads to different reward values obtained by the algorithms in each training round. Therefore, to reflect the true performance

of the algorithms for all the tasks in this domain, we ran four independent trials and reported the average episode reward for SAC-CRM, DDPG-CRM, and Option-DHRM. The lines in the figures below show the average episode reward over those four trials, and the shadowed areas show the range between the highest and the lowest episode reward among those four trials at their specific training steps. We chose to run these algorithms four times because the average reward was pretty stable after four trials; also, since each trial run required up to ten hours of execution time, four trials is what we were able to do during the time period. Additionally, because TD3-CRM and PPO-CRM did not perform well in the initial experiments, we showed one of the typical trials of TD3-CRM and PPO-CRM for each task to show their actual performance over time.

5.1.1 Performance on Task 1 and Task 2

The first two tasks are similar, involving going back and forth between two designated points. When the agent completes a round trip between two endpoints, it will receive a reward value of 1000.

In the first two tasks, because the agent is moving back and forth between two points, its movement route is fixed, which means the agent can take full advantage of the counterfactual experience generated by the CRM. Specifically, in Task 1, since the agent is in two different RM states at point A and point B, after the agent completes a back-and-forth movement between point A and point B, the experience of how to switch from the RM state at point A to the RM state at point B will be stored. The next time when the agent starts from point A, since the task is still to reach point B, which is the same as the previous task, the agent knows what kind of actions it needs to do to transfer the RM state at point A to the RM state at point B as soon as possible so it can reach point B faster. Similarly, when the agent returns to point A from point B, it can also use the previously stored counterfactual experience that can make the RM states change to transfer between RM states, and thus to return to point A as soon as possible.

More importantly, CRM can make the agent learn in parallel. Suppose the agent's current goal is to get to point B and then to point A, but it arrives at point A before getting to point B. By referring to the counterfactual experience, the agent can learn that going backward is disadvantageous when going towards B, but advantageous when going towards A. As such, CRM has already made progress in learning how to get to point A, so as long as the agent gets to point B, it can use the experience of how to get to point A to finish the task more quickly.

In Task 2, the distance between the two endpoints is longer than Task 1. In this task, two advantages of CRM can be exploited. One advantage is that RM can break down a complex task into subtasks, and the other is that CRM provides counterfactual experience for faster learning of subtasks. Specifically, we can

add intermediate points in the middle between the two endpoints and use a reward machine to decompose this task into several subtasks, which will be potentially more straightforward for the agent to learn. Because multiple intermediate points are added in the middle, the agent does not consider the two endpoints as the initial learning goal, but each point between the two endpoints as a subgoal to learn. In this way, although the distance between the endpoints is far, the agent can learn more efficiently by reaching each intermediate point as a subgoal. The agent will record the experience of reaching the next subgoal after completing the previous so that it can use the experience when it has the same subgoal later.

First, to verify that the algorithms can take advantage of the counterfactual experiences provided by CRM, we picked SAC and DDPG as the baseline algorithms and compared their performance with SAC-CRM and DDPG-CRM in the first two tasks. The baseline algorithms run on the cross-product baseline of reward machines. The environment is an MDP, where the states are the cross-product of the environment states and the RM states. As the first two tasks are two of the simplest tasks, and SAC and DDPG are algorithms with good overall performance (which can be seen in the later part of this chapter), we believe that if the baseline SAC and DDPG cannot perform well in the first two tasks, they will not perform well in more complex tasks.

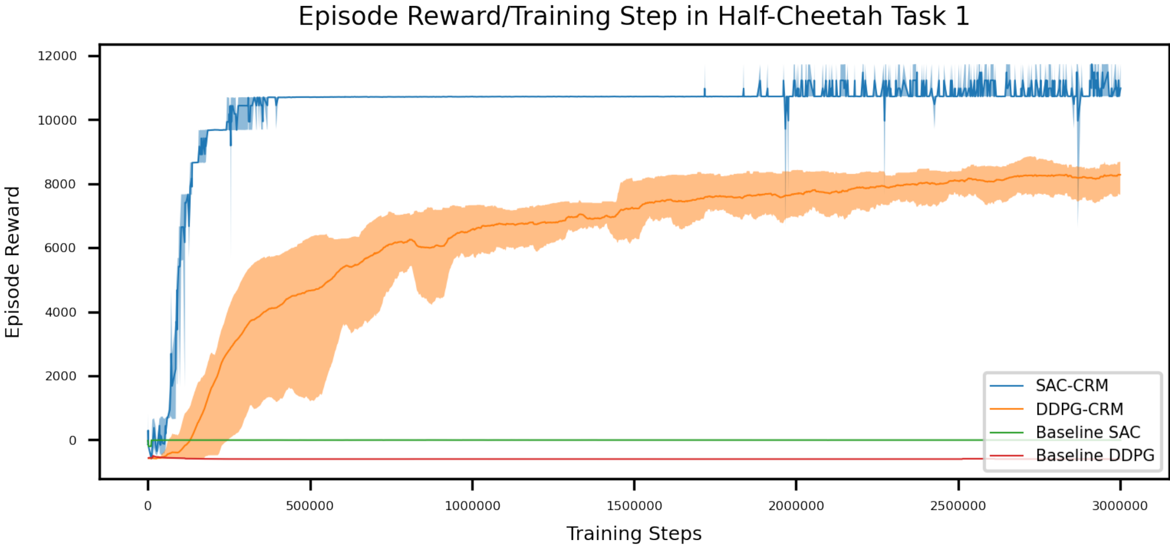

Figure 5.7: CRM vs baselines in Half-Cheetah Task 1

Figure 5.7 and 5.8 show the performance of the baseline SAC and DDPG compared to SAC-CRM and DDPG-CRM, where the horizontal coordinate represents the total number of training steps, which is three million, and the vertical coordinate represents the total reward value received by the agent within an episode (of 1000 training steps). It is evident that in this kind of continuous action domain, the agent cannot learn

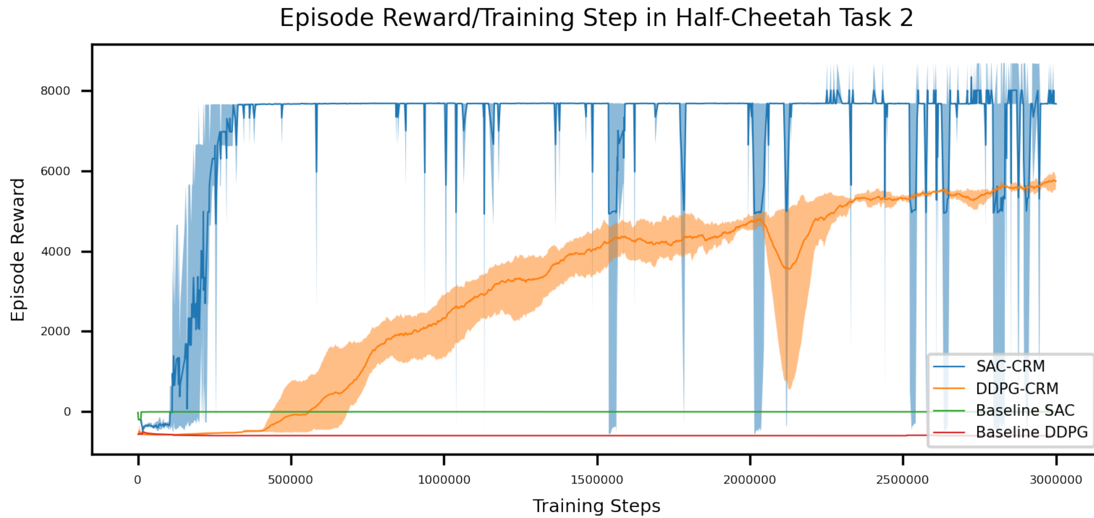

Figure 5.8: CRM vs baselines in Half-Cheetah Task 2

a good policy with just the baseline algorithms, and CRM-based algorithms vastly outperform the baselines in terms of learning speed and reward. This is because CRM can efficiently share counterfactual experiences among different RM states. For example, in Task 1, the agent using CRM can learn to achieve one task while trying to achieve a different one (reaching point A while trying to get to point B), which the baseline algorithms cannot do since the counterfactual experiences are not provided.

As such, in the rest of this chapter, since the baseline algorithms have much worse performance than the RM-based algorithms in the simplest tasks we consider, we will only compare the performance of the RM-based algorithms.

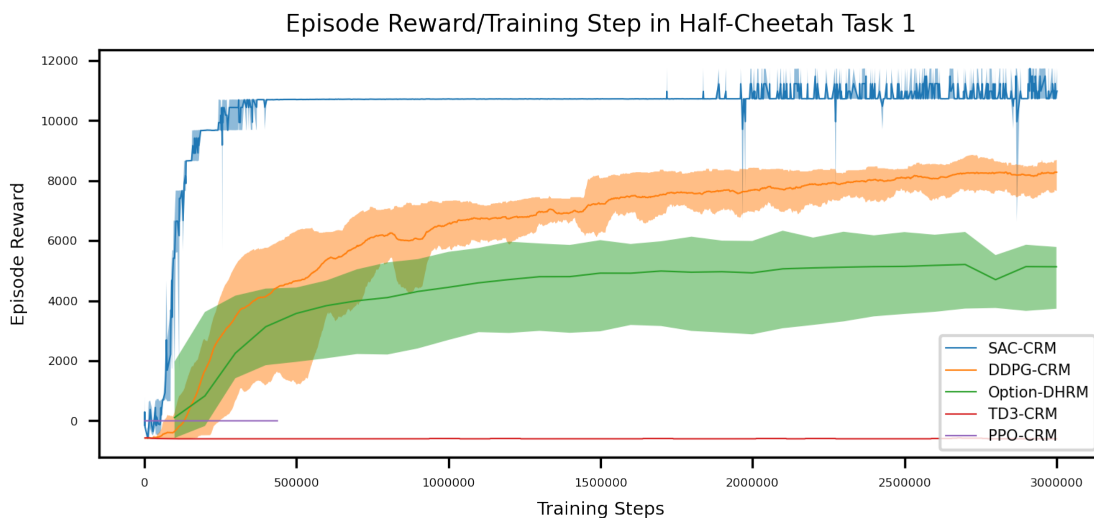

Figure 5.9: Results for Task 1 in the Half-Cheetah domain

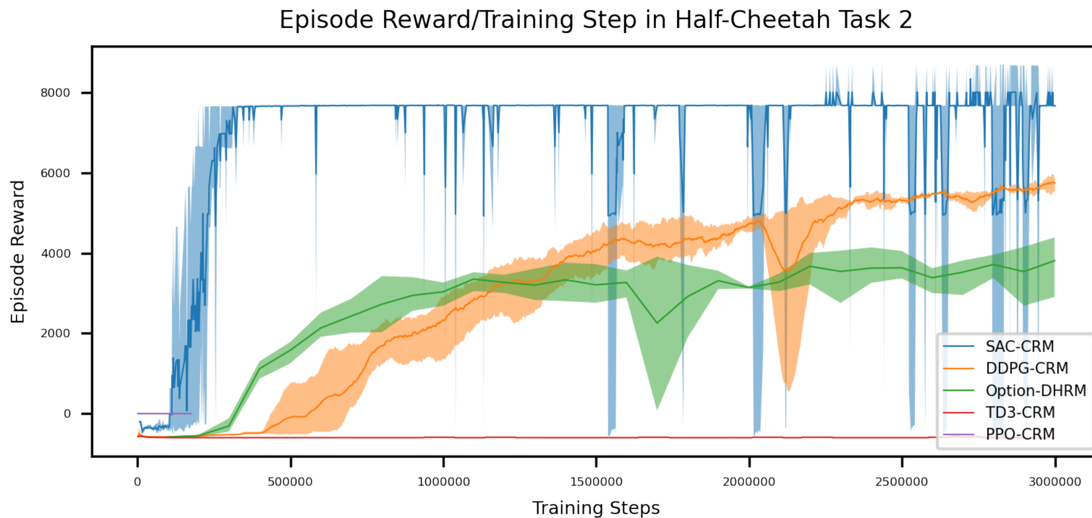

Figure 5.10: Results for Task 2 in the Half-Cheetah domain

Figures 5.9 and 5.10 show the performance of all the proposed and existing RM-based algorithms in Task 1 and Task 2, respectively. The results show that the performance of SAC-CRM is the best among all the algorithms, with a much faster learning speed and a higher reward value than the other algorithms. In terms of learning speed in Task 1, SAC-CRM can achieve the same level of performance as the second-best performer, DDPG-CRM (with its highest episode reward value), in 150,000 training steps, which is up to 20 times faster than DDPG-CRM. In terms of reward, the mean reward value after two million training steps of SAC-CRM is up to 30% higher than DDPG-CRM. It can be seen from the graph that the highest episode reward of SAC-CRM is about 11,000, and considering the fact that the agent will receive a reward value of 1,000 each time it completes the task, which means that the agent trained using SAC-CRM can successfully complete the task about eleven times in one episode, three more than DDPG-CRM. SAC-CRM also performs the best in Task 2, with the fastest learning speed and the highest episode reward.

It can also be seen that the other algorithms do not perform as well as SAC-CRM. In Task 1, DDPG-CRM has the second-best performance, and its learning curve is more stable than SAC-CRM's. However, in Task 2, the learning curve of DDPG-CRM has a noticeable fluctuation during the middle of the learning, and it tends to be stable after 2.4 million steps. Option-DHRM also performs reasonably well, but the rewards it gets are not as high as SAC-CRM and DDPG-CRM. TD3-CRM cannot complete the task within the training period.

In addition, we can find that the curve length of PPO-CRM in the figure is short and does not complete three million steps. This is because PPO-CRM spends a lot of time on sampling and calculating the RM states of the trajectories, resulting in very slow iterations during the training process. In fact, the curves of

PPO-CRM in the figures are the results of ten hours of training. In contrast, all the other algorithms can complete the training of three million steps in less than ten hours. It can be seen that the combination of CRM and PPO is not practical compared to other algorithms in actual training, mainly because of its poor sampling efficiency.

Since the learning speed of PPO-CRM is very slow, we wondered if the PPO algorithm with the cross-product baseline could have better performance. Because the cross-product baseline of PPO does not generate any counterfactual experience, the amount of data it needs to deal with will be significantly less than that of PPO-CRM. Thus, the baseline PPO should have a faster learning speed and potentially learn better policies than PPO-CRM. To evaluate this, we tested the baseline PPO in the first two tasks of the Half-Cheetah domain. Figures 5.11 and 5.12 show the learning curves of PPO-CRM and the baseline PPO algorithm in Tasks 1 and 2, respectively. Since the running time of all algorithms is limited to ten hours, it can be seen from the curves that the learning speed of the baseline PPO is much faster than that of PPO-CRM. In fact, the curve of the baseline PPO in Task 1 is the result of running for less than 30 minutes. For each hour, the baseline PPO can complete more than 5 million learning steps; in contrast, PPO-CRM can only complete less than 0.5 million steps after ten hours of learning. The situation is similar in Task 2, where the baseline PPO can complete nearly 5 million steps in an hour, while PPO-CRM can only complete less than 0.25 million steps in ten hours.

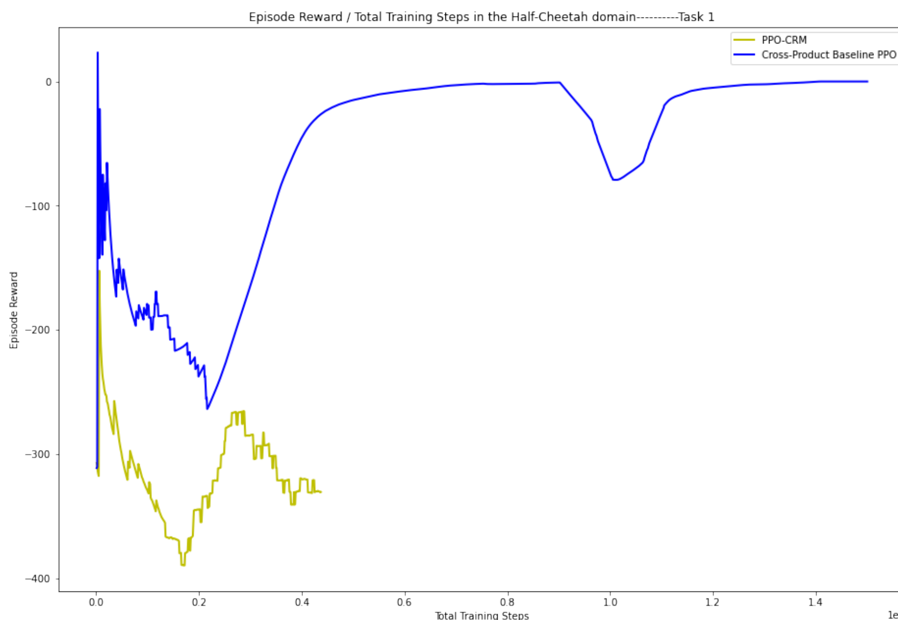

Figure 5.11: PPO-CRM vs baseline PPO in Half-Cheetah Task 1

In terms of rewards, the baseline PPO can get slightly better rewards than PPO-CRM in the first two tasks, but it is not ideal overall. The policies learned by the baseline PPO improve faster than those of

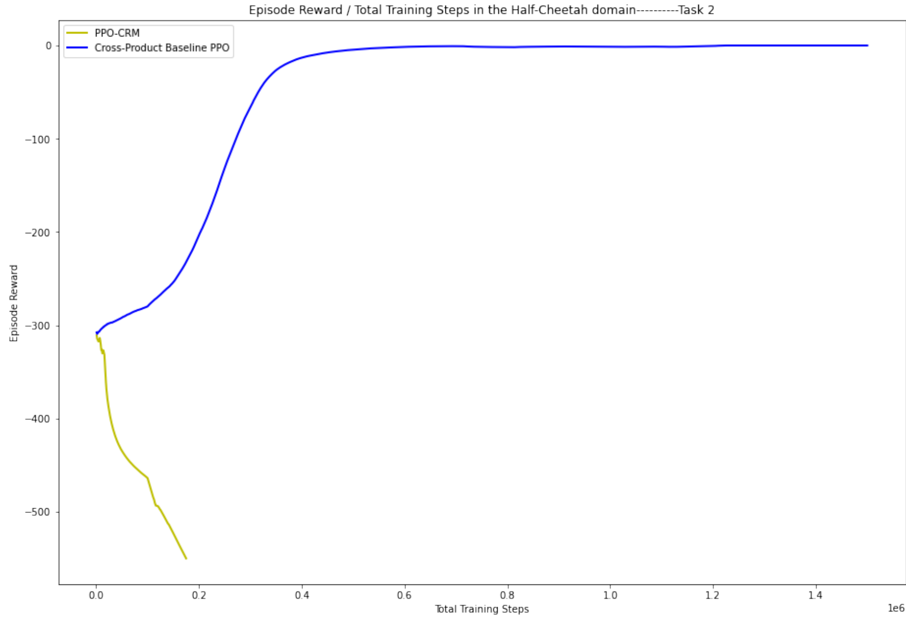

Figure 5.12: PPO-CRM vs baseline PPO in Half-Cheetah Task 2

PPO-CRM, but similarly to the baseline SAC and baseline DDPG, they don't really improve over time.

To summarize, the baseline PPO learns quickly but not well, at least in these Half-Cheetah domain tasks. When the results of the baseline SAC and baseline DDPG are taken into account, it can be seen that CRM is still an indispensable part if the agent wants to learn a good policy in an MDPRM.

5.1.2 Performance on Task 3

Task 3 adds the operation of making the agent fold back midway, which is more complex compared to the first two tasks. Figure 5.13 shows the performance of all the algorithms in Task 3. In this task, SAC-CRM still performs well, with higher reward values and faster learning speeds than other algorithms. It can be recognised that SAC-CRM performs well in the first three tasks, indicating that SAC-CRM would be the best choice for tasks where the agent needs to move back and forth between endpoints and with some foldback operations.

In addition, it can be seen that the performance of TD3-CRM and PPO-CRM is still much worse compared to the other algorithms. As with the first two tasks, PPO-CRM still has a very slow learning speed.

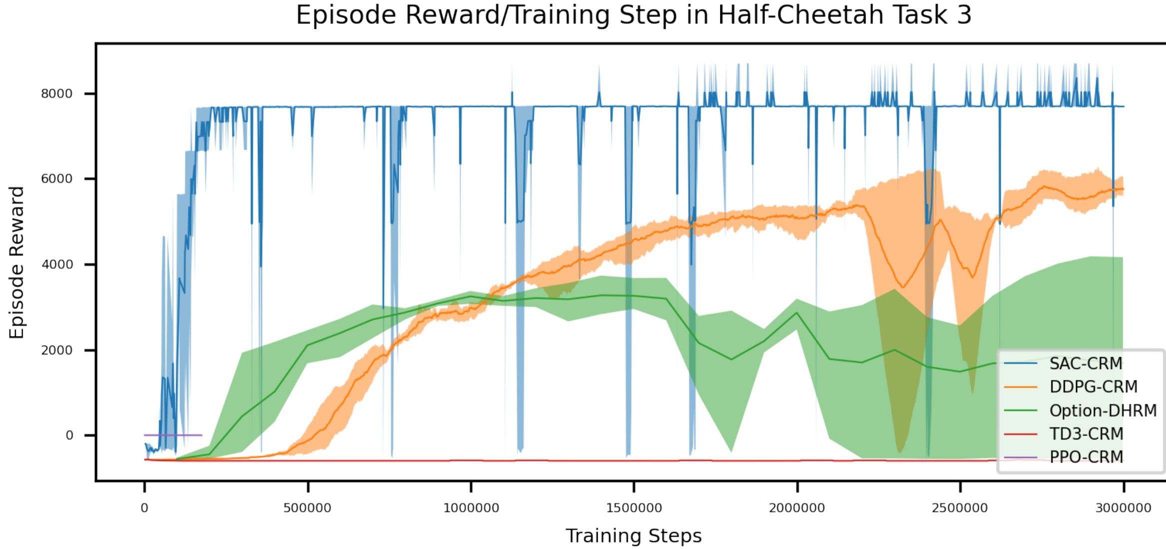

Figure 5.13: Results for Task 3 in the Half-Cheetah domain

5.1.3 Performance on Task 4

Figure 5.14 shows the performance of all the algorithms in Task 4. SAC-CRM still has the best performance in this task, and its lead is larger than in the other tasks. DDPG-CRM still performs reasonably well, but its learning speed in the early period is significantly slower than SAC-CRM. Option-DHRM is also able to maintain the same level of performance as DDPG-CRM in this task, but the performance gap between Option-DHRM and SAC-CRM becomes larger compared to that in the previous tasks. TD3-CRM and PPO-CRM still don't perform well, and PPO-CRM still has a very slow learning speed.

According to the experimental data of the first four tasks, it is obvious that SAC-CRM is the best performer among all the algorithms. There are several reasons for the excellent performance of SAC-CRM. The first reason is that the SAC algorithm is more exploratory than the DDPG algorithm because of the incorporation of maximum entropy. Because the agent's control is continuous in the Half-Cheetah environment, the agent must have precise control over each joint and part of its limbs in order to move in the target direction. In other words, the higher the degree of cooperation between each part, the faster the movement speed of the agent. Compared with SAC-CRM, the agent trained by DDPG-CRM and Option-DHRM can complete the task with a reasonable reward. However, DDPG-CRM and Option-DHRM explore less than SAC-CRM, and the algorithm's understanding of the optimal solution will fall into a local optimum too early. In other words, DDPG-CRM and Option-DHRM can make the limbs of the agent cooperate to complete the movement, but the degree of cooperation between the limbs is not optimal, so that the agent moves at a slower speed. In contrast, SAC-CRM is more exploratory, so each joint can try more types of

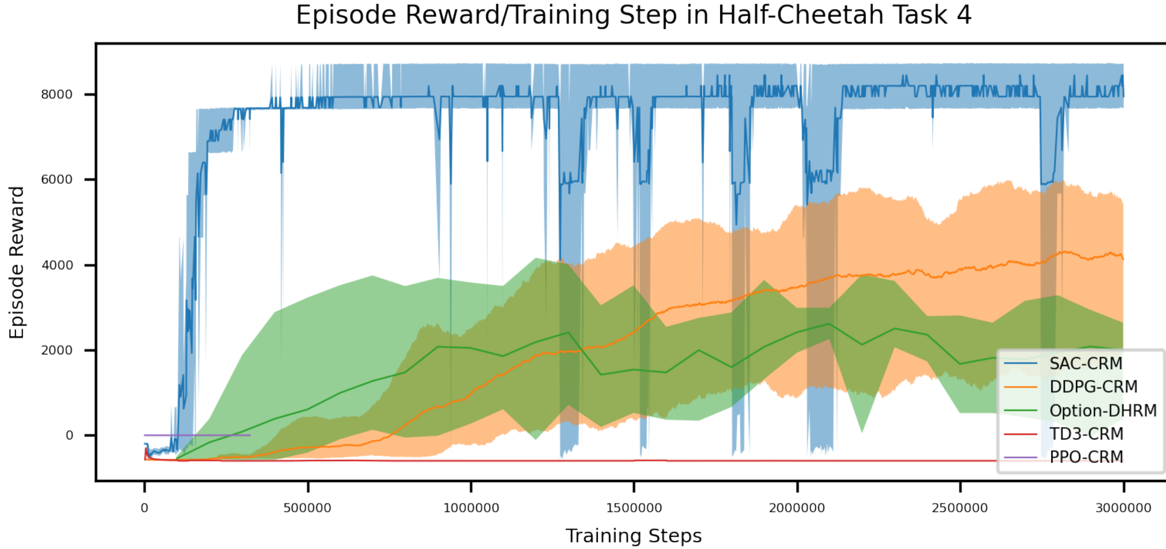

Figure 5.14: Results for Task 4 in the Half-Cheetah domain

cooperation to make the agent move faster and get a higher reward.

Meanwhile, let us take a closer look at the SAC-CRM learning process. According to the learning curve, even when it already reaches a high reward value, we can see that it sometimes produces obvious fluctuations in the reward value obtained by each episode. The reason for the fluctuations is that in SAC, because of the inclusion of maximum entropy, even if the agent can achieve a high reward value with the current policy, SAC will still encourage the agent to continue exploring by maximising the entropy and keeping trying new actions. The later part of the curve with significant declines indicates that not all new actions tried by the agent have positive effects, and sometimes inappropriate actions can seriously reduce the reward value. However, the continuous exploration of new actions by SAC is the key factor leading to a more coordinated performance of the agent's limbs and a higher average reward.

5.1.4 Performance on Task 5

Figure 5.15 shows the performance of all algorithms in Task 5. It can be seen that although SAC-CRM performs better than TD3-CRM and PPO-CRM, it does not often reach F, resulting in low reward values. SAC-CRM fails to perform well on this task for two main reasons. First, CRM does not play a prominent role in this task. In the previous tasks, there were repetitive movement routes, so the agent could take advantage of the counterfactual experience provided by CRM and could learn quickly when encountering a previously travelled route. However, in the fifth task, the agent has only one unique movement direction and does not have any backward movement. As a result of not taking any previously traveled routes in this task, the agent

does not experience the same RM states as previously during the learning process. Moreover, the state space it traverses is different for each subtask. Thus, the counterfactual experience provided by CRM has little effect on this task. The second reason is the SAC algorithm's entropy maximisation property. As mentioned before, SAC is highly exploratory; it can continuously make the agent try to move back and forth within a short distance, which was needed in the previous tasks. However, in the fifth task, the agent's movement direction is fixed, so the high exploratory nature of SAC has a negative impact in this task, which causes the agent to make many movements that are opposite to the target direction while exploring, resulting in a significantly lower probability of the agent making the correct movement, and thus a significantly lower quality of the policy learned and the average reward. As demonstrated by this task, the highly exploratory nature of new actions is not always advantageous, and it can even be detrimental when dealing with tasks that require monotone actions.

Figure 5.15: Results for Task 5 in the Half-Cheetah domain

In contrast, DDPG-CRM performs better in this task. The reason is that the exploratory nature of DDPG-CRM is relatively weak among these algorithms, and the agent can determine the action of moving in the target direction earlier and keep exploiting this kind of action. During the early learning phase, when the agent happens to reach point F and receive the reward, DDPG-CRM will assume that a single type of movement in the target direction is enough to receive the reward and will reduce the exploration of other movements in the opposite target direction. This extensive use of existing actions will assist the agent in receiving higher reward values for this task. So, the low rate of exploration and high exploitation of actions are good for this task; they can help the agent get a higher reward value.

Among the other algorithms, it can be seen that the most suitable algorithm for learning this task is

Option-DHRM. Option-DHRM is ahead of the other algorithms both in terms of learning speed and final reward value. As mentioned earlier, Option-DHRM can quickly reach the local optimal solution with the help of options. In this task, since the agent's goal is to keep moving in one direction, each intermediate point reached by the agent is equivalent to reaching a local optimal goal. The local optimal option in this task is very close to the global optimal solution; that is, the local optimal policies learned by Option-DHRM are actually the global optimal policies. Therefore, the agent in this task with Option-DHRM can derive the global optimal solution very quickly.

The performance of TD3-CRM is still not good in this task; the reward values it gets are the lowest among all the five tasks by far. Similar to its performance in the previous tasks, PPO-CRM continued to learn very inefficiently in this task, and after more than ten hours of learning, PPO-CRM is still unable to complete the required number of learning steps. Moreover, it can be observed that the reward value achieved by all CRM algorithms in this task is relatively low compared to the previous tasks. The main reason for this is that CRM's counterfactual experience in this task cannot assist the agent in reducing exploration. Moreover, the distance between the starting point and the endpoint in this task is the longest, so the reward in this task is the most sparse compared to the other tasks. The sparsity of the reward will lower the chance that the agent completes the task and increase the workload of the exploration, thus decreasing the agent's learning efficiency.

5.2 Results in the Ant Domain

To further increase the complexity of the environment and tasks, we also tested the performance of all the algorithms in the Ant domain. The Ant robot is a three-dimensional robot with one torso (a free rotational body) and four legs (each with two links). By applying torques to the eight hinges connecting the two links of each leg and the torso, the primary goal of the Ant robot is to coordinate the four legs to move in any direction on the plane. An abstract representation of the Ant domain is shown in Figure 5.16.

There are reasons to choose the Ant domain. First, the Ant domain is a 3D environment, and the agent will have a larger moving space, which means that this environment can make the agent produce more types of states. Second, the Ant robot has more joints and thus requires more complex movement coordination mechanisms to make the body move, making learning the movements more difficult. In this domain, all the algorithms will face more complex motion characteristics, environmental states, and greater learning pressure. Thus, the Ant domain is ideal for testing how well the RM-based algorithms can perform in a more complex environment.

We will test three tasks in the Ant environment. In all three tasks, the ant robot's starting point will be

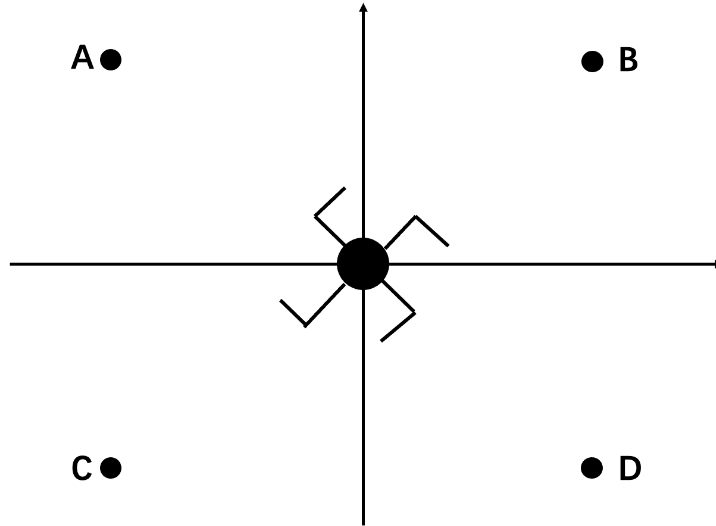

Figure 5.16: The Ant domain

a random location near the origin. The tasks are shown in Table 5.2. Task 1 is similar to the first task in the Half-Cheetah domain, in which the robot starts from the starting point, reaches point B, then point A, and then repeatedly moves back and forth between point A and point B. Task 2 is also like the second task in the Half-Cheetah domain, which is a more complex version of Task 1. Note that in the Ant domain, points A, B, and C are not along a straight line, which will make the task more complex than in the Half-cheetah domain. Also, the agent's starting point in this domain is an arbitrary location near the origin, which is different from that in the Half-Cheetah. Finally, to try to reach the limit of these algorithms, we designed Task 3 with the most constraints in all environments. First, the agent can choose one point among point A

Table 5.2: Tasks for the Ant domain.

Task No.	**Description**
1	Start from an arbitrary position around the origin, first go to point B, then repeatedly go back and forth between A and B.
2	Start from an arbitrary position around the origin, first go to point A, then to point B then to point C, and then back to point B and then to point A and then repeat indefinitely.
3	Start from an arbitrary position around the origin, choose either point A or point B, and go to the chosen point, then go to point C, and then to point D. From point D, if it chose to go to point A before, it must return to point A; if it chose to go to point B before, it must return to point B. After reaching the chosen point, stop.

and point B and move there, then move to point C, then move to point D. From D, if it chose to move to point A before, then it must return to point A. If it chose to move to point B before, then it must return to point B. The RM automaton of Task 3 is shown in Figure 5.17. In this domain, we will also show the average and max/min episode reward in four independent trials for SAC-CRM, DDPG-CRM, and Option-DHRM. The line and shadow colours for all the algorithms are the same as in the Half-Cheetah domain.

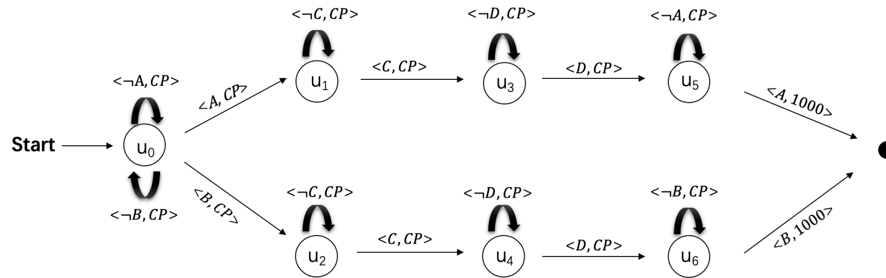

Figure 5.17: The RM automaton of Task 3 in the Ant domain.

5.2.1 Performance on Task 1 and Task 2

Figures 5.18 and 5.19 show the learning curves of all the algorithms in the first two tasks, respectively. Only SAC-CRM and DDPG-CRM can obtain significant rewards in the first two tasks within the specified number of learning steps, while the other algorithms cannot.

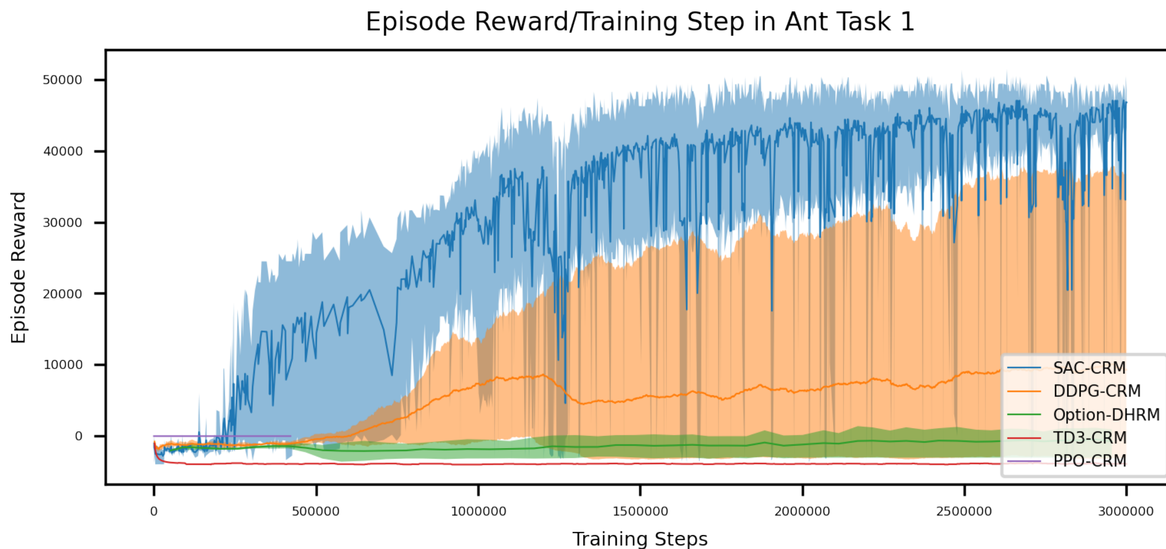

Figure 5.18: Results for Task 1 in the Ant domain

According to the learning curves, it can be seen that SAC-CRM still performs the best among all the

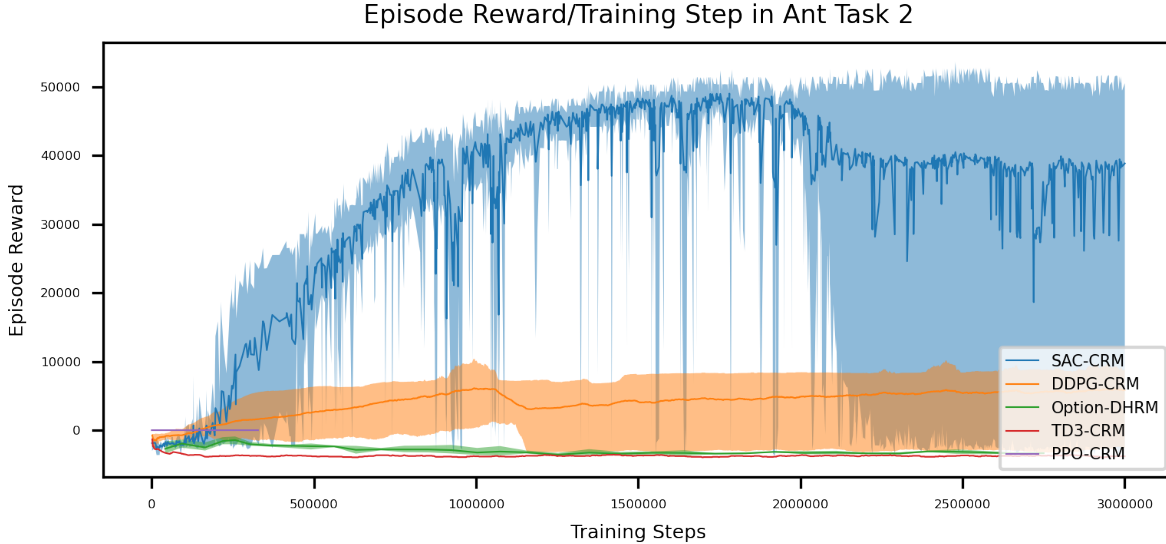

Figure 5.19: Results for Task 2 in the Ant domain

algorithms. Compared to DDPG-CRM, SAC-CRM has a significant lead in learning speed and reward values. The main reason for this is the higher exploration feature of the SAC algorithm. In the Ant domain, the agent's movement changed from backward and forward in Half-Cheetah to backward, forward, left, and right, which means a larger movement space. The high exploratory tendency of SAC plays an essential role in trying new directions in the environment and exploring the coordination between joints, which allows the agent to try more different directions and explore the coordination between limbs, thus improving the movement speed faster than the other algorithms. DDPG-CRM, on the other hand, slows down its exploration rate as it learns, so its performance improves slowly.

We can also draw other conclusions about SAC-CRM from the learning curves and the average episode rewards. First, the figures of the first two tasks show that the curves of SAC-CRM fluctuate more compared to the similar tasks in the Half-Cheetah domain. Moreover, the difference between the maximum reward value and the average episode reward value after two million steps is larger, which indicates that the policy obtained by SAC-CRM is still not close to optimal, even at the later stage of learning. The main reason is that the Ant domain is a complex environment, and the diversity of environmental states and actions puts pressure on the agent's learning.

It is noteworthy that Option-DHRM performs poorly in this domain. Even when the tasks are similar, the performance of Option-DHRM in the Ant domain has a large gap compared to that in the Half-Cheetah domain. As previously stated, Option-DHRM is good at quickly finding a local optimal solution, but its shortcoming is that policies learned through options easily fall into the local optimal. In the Half-Cheetah

domain, because the environmental complexity is lower than in the Ant domain, the agent can still rely on the local optimal policy to achieve relatively high reward values. However, the gap between local optimal policy and global optimal policy becomes bigger in more complex environments; in a more complex environment, the local optimal policy will not be sufficient for the agent to get a high reward, which is the direct reason for the poor performance of Option-DHRM in the Ant domain. For TD3-CRM, it can be seen that its performance is similar to that in the Half-Cheetah domain, and it still performs poorly. Also, PPO-CRM still suffers from a very long learning time and underperforms.

5.2.2 Performance on Task 3

Task 3 adds the most constraints to the agent's movement. In this task, the agent not only has to select to move to the target points as soon as possible but also needs to remember the points it has previously selected and perform precise folding operations to get back to the selected points in the subsequent task process. The complexity of this task further increases the learning difficulty for the agent.

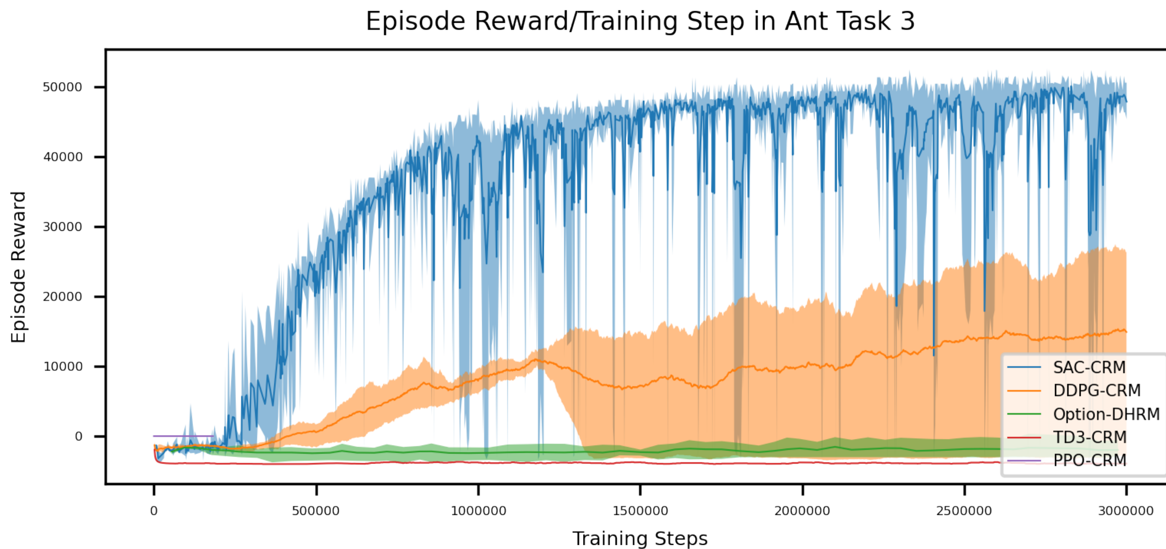

Figure 5.20: Results for Task 3 in the Ant domain

The results in Figure 5.20 show that SAC-CRM is able to maintain a relatively high reward value most of the time even though the task becomes more complex, which shows that SAC-CRM is still the best choice for the tasks in the Ant domain. Although the learning curve of DDPG-CRM fluctuates less, it does not have the same level of performance as SAC-CRM in terms of reward. TD3-CRM and PPO-CRM still do not perform well. Option-DHRM in this task also has similar performance to the first two tasks, which is not ideal.

5.3 Discussion

5.3.1 Overall Performance of SAC-CRM

From the experimental results, it can be seen that SAC-CRM has the best performance in most of the tasks. Except for Task 5 in the Half-cheetah domain, SAC-CRM outperforms other RM-based algorithms, both in terms of the learning speed and the quality of the policy. The lead of SAC-CRM is significant in the Half-cheetah domain; in the more complex Ant domain, SAC-CRM's lead is more even pronounced. It can be concluded that SAC-CRM is the best choice for agents to learn most types of tasks in both the Half-Cheetah and Ant domains.

SAC-CRM has such a lead mainly because of its unique entropy-based policy update mechanism. Since the experimental environments are continuous, there will be a considerable number of action combinations that can make the agent move. Among them, some combinations will make the agent fail; some will make the agent move in the desired direction but at a slow speed; others will make the agent move at a faster speed. As such, allowing the agent to find the most appropriate combination of actions for moving in each direction will be the key to influencing the speed and efficiency of the agent's movement and thus the reward it can get. The consideration of the entropy value makes SAC-CRM more exploratory. It allows the policy to be as random as possible to avoid it falling into a local optimum early. Thus, it can ensure that the agent can keep a high degree of exploration while trying to maximize the reward value.

Another key to the success of SAC-CRM is that the type of policy it generates is a stochastic policy. In seeking optimal actions, especially in the continuous action domain, there is often more than one optimal action. For example, the Cheetah robot in the experiments can achieve the same speed through various kinds of cooperation of limbs. In a deterministic policy, the output of an action combination is unique, which limits the possibility for the agent to discover better action combinations. In contrast, SAC-CRM learns a stochastic policy; in each state, instead of outputting only one action combination, the policy will save all the available actions in this state and set the probabilities of these actions according to their Q-values. As such, when the agent is in the same state next time, it will no longer be limited to selecting only the optimal action of the current state (with a high probability of being locally optimal), but will have a chance to select some of the sub-optimal actions (which may be globally optimal) and thus have a chance to find a better combination of actions. While SAC-CRM has a broader "vision" in action selection, different combinations of actions can be tried frequently, resulting in faster learning for problems such as controlling robots that require multi-limb coordination.

However, SAC-CRM does not always perform well. In Half-Cheetah Task 5, it can be seen that the policy that SAC-CRM learned is not on the same level as DDPG-CRM and Option-DHRM. Therefore,

the conclusion that can be drawn is that when the counterfactual experiences of reward machines can be effectively utilized, the policies learned by SAC-CRM are more dominant because there is some overlap in the relevant action spaces for different RM states; the robots trained by SAC-CRM can have faster movement speed and thus get higher reward values within the specified number of steps.

5.3.2 Overall Performance of DDPG-CRM

Although the rewards that DDPG-CRM can get in all the tasks are not the highest, its performance is more consistent compared to the other RM-based algorithms. In the Half-Cheetah domain, SAC-CRM performs the best in the first four tasks; however, it does not do well in the fifth task. In the fifth task, although Option-DHRM performs the best both in terms of the learning speed and the reward it gets, DDPG-CRM's performance is still at the top tier level. When it comes to the Ant domain, Option-DHRM falls back in performance, while DDPG-CRM can still maintain a reasonable level of performance, and its learning curves are usually very stable.

The weakness of DDPG-CRM is its slow learning speed, and the relatively low reward it can get. As shown in the experimental results, in most of the tasks, DDPG-CRM performs worse than SAC-CRM, both in terms of the learning speed and the rewards it gets. Moreover, when facing the more complex Ant domain, the gap between DDPG-CRM and SAC-CRM has been further increased. Compared to SAC-CRM, DDPG-CRM is less exploratory because it learns a deterministic policy, which limits the agent in seeking better action combinations. Also, DDPG-CRM still learns based on a traditional Q-function, which can over-estimate the Q-values and result in the agent falling into a local optimum too early.

Overall, DDPG-CRM is a robust algorithm since it has consistent performance across all the tasks. For a relatively simple environment, such as Half-Cheetah, DDPG-CRM may be the best choice if robustness is more important than optimal performance. It can handle almost all types of tasks and has relatively stable performance. However, the rewards that DDPG-CRM offers are not the best; when the environment becomes complex, DDPG-CRM cannot achieve the same level of reward as SAC-CRM, so it is no longer an appropriate choice for most tasks in complex environments.

5.3.3 Overall Performance of Option-DHRM

From the experimental results, it can be seen that Option-DHRM performs reasonably well in the Half-cheetah domain. In the first four tasks, although Option-DHRM is not as good as SAC-CRM, it can still achieve the same level of performance as DDPG-CRM, both in terms of learning speed and reward. Moreover, Option-DHRM performs exceptionally well in the fifth task in the Half-Cheetah domain; its learning speed

and reward are the best among all the algorithms.

However, once the environment is switched to the Ant domain, the performance of Option-DHRM falls drastically. This is because Option-DHRM is only good at finding local optimums, but the difference between the local optimum and the global optimum becomes larger when the environment and the tasks become more complex, so Option-DHRM loses its advantages when facing a complex environment. Therefore, in practical applications, Option-DHRM is a good algorithm for environments with low complexity. Although the final policy may not be the global optimum, Option-DHRM is capable of dealing with various types of tasks, especially tasks with sparse rewards. However, in more complex environments, the advantages of Option-DHRM will no longer apply, and it may no longer be a usable algorithm.

5.3.4 Overall Performance of TD3-CRM

For TD3-CRM, it can be seen that its learning performance in all the tasks is far from ideal, and the rewards it gets are the lowest among all the algorithms. The reason is the conflict between CRM and the policy updating mechanism of TD3.

Firstly, TD3-CRM learns two Q-functions for an action and always assigns the lower Q-value to the action. This is reasonable for the pure TD3 in an MDP because the Q-values in an MDP are usually overestimated. However, this is not the case in an MDPRM. In an MDPRM, the RM information is critical for the agent to transfer from one RM state to another; specifically, the actions that can make the RM state change usually have high Q-values, which encourage the agent to keep using these actions to make transitions between the RM states. Nevertheless, TD3 always tries to "underestimate" the Q-values, which will avoid these beneficial actions.

Moreover, the target policy smoothing regularization of TD3 also restricts the agent from choosing the most appropriate actions. Because of the added noise to the actions, TD3-CRM will not choose the best actions that can cause the RM state changes. Instead, it always chooses the actions that are close to the best ones. On the one hand, CRM always encourages the agent to perform the best actions so far that can cause the RM state to change; on the other hand, TD3 always discourages the agent from doing so, which will confuse the agent. As a result, the agent can rarely perform the right actions, which leads to its poor performance.

5.3.5 Overall Performance of PPO-CRM

PPO-CRM is unique among all the algorithms. PPO is an on-policy algorithm; it learns through sampled trajectories, and the previous experience will not be stored in the process of learning the policies, which

makes it difficult to exploit the benefits of CRM. Meanwhile, if PPO wants to learn in an MDPRM, it must first sample a very large number of trajectories and then, as the learning progresses, calculate the corresponding RM states and RM rewards and place them into the trajectories. This series of operations includes a lot of data input and output, which is time-consuming and leads to poor learning efficiency. It can be seen through the experiments that the PPO-CRM has a very slow learning speed compared to our other algorithms, and it shows that PPO-CRM is not a good choice in an MDPRM. Through the experiments, it can be seen that it is tough to combine CRM with such an on-policy algorithm to produce good results, and the combination with off-policy algorithms will be the better choice for using CRM.

Chapter 6

Conclusion

The development of AI has provided many conveniences to human beings' daily lives. AI technologies such as machine automation have significantly improved productivity and will play an indispensable role in future human life. In this thesis, we use deep reinforcement learning to train intelligent agents and define various types of tasks to simulate real-world application scenarios that intelligent robots might encounter.

Training a usable deep RL agent for a specific scenario usually requires a large amount of training data and a long training time. Moreover, the environment in which the agent is placed is usually partially observable — as such, trying to learn high-quality policies from the limited information is a major challenge for deep RL. Therefore, observing more information from the environment and learning how to fully use this information will be the key to improving the training efficiency of an agent. Based on these challenges, we have done some work on this thesis.

In order to observe more valuable data from the environment, we applied the framework of reward machines to the underlying environment. A reward machine can convey the high-level idea of a task to the agent, so that the agent has a deeper understanding of the reward function. For information utilization, we investigate some of the current state-of-the-art deep RL algorithms and combine these algorithms with reward machines to help the agent learn more efficiently.

The research in this thesis is inspired by previous work on reward machines and deep RL algorithms, especially the work of Toro Icarte et al. [23]. Based on the related work, we make the following contributions.

6.1 Summary of Contributions

The contributions of the thesis are summarized in this section. In Chapter 3, we discussed three popular recently proposed deep RL algorithms: SAC, TD3, and PPO. Then in Chapter 4, we combined the reward

machine model with several current mainstream deep RL algorithms and proposed three new algorithms based on reward machines, called SAC-CRM, TD3-CRM, and PPO-CRM, respectively. We also showed how to combine typical deep RL algorithms with reward machines and how to incorporate counterfactual experiences into the algorithms. In Chapter 5, in order to simulate the tasks that an intelligent agent might encounter in the real world, we defined a total of eight different types of tasks in two simulated continuous action domains, including two previously existing tasks from the work of Toro Icarte et al. [25] and six brand new tasks. On this basis, we tested the performance of all RM-based deep RL algorithms in eight tasks and found that the newly proposed SAC-CRM performed the best in most of the tasks. Last but not least, we compare the performance of all the RM-based deep RL algorithms, analyze their advantages and disadvantages for different task types, and recommend different algorithms for different application scenarios.

6.2 Future Work

In this thesis, we propose three novel algorithms by combining reward machines with some of the most cutting-edge deep RL algorithms. Meanwhile, we test and analyze the performance of all the RM-based deep RL algorithms in two different continuous action domains with a total of eight tasks. However, we have not explored the full potential of these algorithms. In this section, we propose several aspects that could be improved in the future.

1. **Additional Parameter Tuning**

   A deep RL algorithm typically contains dozens of parameters; each parameter can affect the algorithm's performance. For each specific task, it is usually necessary to perform a large number of tuning and testing operations on an algorithm to bring out its best performance. Because there are various task types and algorithms in this thesis and the time constraints, for all the algorithms, we use uniform parameters in the testing of all the tasks, which may not allow each algorithm to perform at its best. In subsequent work, additional tuning of the parameters for each task may further improve the algorithms' performance.

2. **More Experimental Evaluation**

   The test environment in this thesis contains two continuous action domains and a total of eight tasks. However, these domains and tasks can represent only a small portion of application scenarios. In other words, some algorithms may perform well in the specific tasks and domains included in this thesis, but this does not mean that these algorithms can still maintain good performance in other tasks or domains. Therefore, adding more domains with more task types in subsequent work can test the

robustness of the algorithms more comprehensively. This would also help us understand better what environment and task characteristics are most suitable for each algorithm.

3. **Studying Ways of Ensuring that SAC-CRM Produces More Stable and Robust Policies**

   From the experimental results, it can be seen that SAC-CRM has the best performance among all the existing RM-based algorithms in most of the environments and tasks, both in terms of learning speed and reward. However, the learning curve of SAC-CRM fluctuates considerably in some tasks. Even at the late stage of learning, the policies of SAC-CRM remain unstable compared to those of other algorithms. Therefore, if we can find ways to make the policies of SAC-CRM more stable, then the robustness of SAC-CRM can be brought to a higher level and make it a more reliable algorithm.

4. **Studying Ways of Improving the Learning Efficiency of PPO-CRM**

   In this thesis, we combine the PPO algorithm, which is one of the typical on-policy algorithms, with CRM. However, the learning speed of PPO-CRM is very slow due to the addition of the counterfactual trajectories. The way we combine PPO and CRM is relatively brute force, and the results show that our approach can bring little help to the baseline PPO and will significantly drag down the learning efficiency. Thus, in future work, one can examine what can be done to optimize the learning mechanism of PPO-CRM.

5. **Exploiting a Reward Machine with an On-policy Algorithm**

   In this thesis, we try to adapt CRM with the PPO algorithm, but the experimental results of our approach are not ideal, and we have not found a way to adapt CRM with such an on-policy algorithm yet. Maybe one approach that could be tried is to adapt CRM with some version of SARSA [27], which is an algorithm that can be used in the discrete action domain, and it updates the policy based on a single action experience rather than a whole trajectory. In any case, how to use a RM to learn faster with an on-policy algorithm still remains an open problem.

6. **Adding Reward Shaping to the CRM-based Algorithms**

   In all of our experiments, we use reward machines to break a complex task into several subtasks to let the agent learn phase by phase. However, the agent only receives the reward when it completes the whole task; it does not receive any reward when it completes any of the subtasks. Hence, the reward for a complex task is very sparse. In order to make the reward less sparse, Toro Icarte et al. also proposed a method called *Automated Reward Shaping (RS)* [16], which provides some intermediate rewards in some subtasks as the agent gets closer to completing the task. This method can encourage the agent

to make progress towards solving the task. Toro Icarte et al. have exploited RS in discrete domains and achieved some better results [25]. In future work, we can also try to add the RS mechanism to the existing CRM-based algorithms to potentially further improve the learning speed.

7. **Applying the RL algorithms to real-world hybrid domains**

   Many real-world control problems are in a hybrid domain [14], which involves both discrete decision variables and continuous decision variables. For example, in autonomous driving, discrete variables include the actions of moving forward or backward, whereas continuous variables include speed and turning angle. However, fully continuous or fully discrete action domains are frequently used to approximate the corresponding optimal control or reinforcement learning problem. These modifications try to adapt the problem to a specific algorithm, which will only be able to handle a single kind of action domain. In this thesis, the RM-based algorithms, such as SAC-CRM and DDPG-CRM, are only able to solve continuous control problems. So in the future, if there were ways to apply these algorithms to hybrid domains, it will have a more practical impact on solving real-world problems.

6.3 Concluding Remarks

In this thesis, we addressed two long-standing challenges in deep RL: partial observability and learning efficiency. Our main contribution consists of combining the current mainstream deep RL algorithms with reward machines. We also defined various types of tasks in different continuous action domains to simulate real-world robot control problems. Our experimental evaluation of these algorithms on these tasks shows tremendous potential use cases that RL may have in the future and provides additional references for the future development of AI robot control.

Bibliography

[1] David Bachman. *Advanced Calculus Demystified*. McGraw-Hill, 2007.

[2] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Trans. Syst. Man Cybern.*, 13(5):834–846, 1983.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[4] Stephan Bütikofer, Diethard Klatte, and Bernd Kummer. On second-order Taylor expansion of critical values. *Kybernetika*, 46(3):472–487, 2010.

[5] Robert Dadashi, Léonard Hussenot, Damien Vincent, Sertan Girgin, Anton Raichuk, Matthieu Geist, and Olivier Pietquin. Continuous control with action quantization from demonstrations. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 4537–4557. PMLR, 2022.

[6] A. Feinberg. Markov Decision Processes: Discrete Stochastic Dynamic Programming (Martin L. Puterman). *SIAM Rev.*, 38(4):689, 1996.

[7] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1582–1591. PMLR, 2018.

[8] Giuseppe De Giacomo and Moshe Y. Vardi. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 854–860. IJCAI/AAAI, 2013.

[9] Antonio Guillen-Perez and María-Dolores Cano. Learning from oracle demonstrations - A new approach to develop autonomous intersection management control algorithms based on multiagent deep reinforcement learning. *IEEE Access*, 10:53601–53613, 2022.

[10] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1856–1865. PMLR, 2018.

[11] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic Algorithms and Applications. *CoRR*, abs/1812.05905, 2018.

[12] Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, S. M. Ali Eslami, Martin A. Riedmiller, and David Silver. Emergence of Locomotion Behaviours in Rich Environments. *CoRR*, abs/1707.02286, 2017.

[13] Richard M. Karp, Michael Luby, and Neal Madras. Monte-Carlo Approximation Algorithms for Enumeration Problems. *J. Algorithms*, 10(3):429–448, 1989.

[14] Boyan Li, Hongyao Tang, Yan Zheng, Jianye Hao, Pengyi Li, Zhen Wang, Zhaopeng Meng, and Li Wang. Hyar: Addressing discrete-continuous action reinforcement learning via hybrid action representation. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[15] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[16] Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In Ivan Bratko and Saso Dzeroski, editors, *Proceedings of the Sixteenth International Conference on Machine Learning (ICML 1999), Bled, Slovenia, June 27 - 30, 1999*, pages 278–287. Morgan Kaufmann, 1999.

[17] John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. Trust Region Policy Optimization. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd Interna-*

tional Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1889–1897. JMLR.org, 2015.

[18] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *CoRR*, abs/1707.06347, 2017.

[19] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller, editors, *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*, pages 1057–1063. The MIT Press, 1999.

[20] Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artif. Intell.*, 112(1-2):181–211, 1999.

[21] Haoran Tang and Tuomas Haarnoja. Learning diverse skills via maximum entropy deep reinforcement learning. Unpublished manuscript available at: https://bair.berkeley.edu/blog/2017/10/06/soft-q-learning/, Berkeley AI Research, Oct 2017.

[22] Yunhao Tang and Shipra Agrawal. Discretizing continuous action space for on-policy optimization. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 5981–5988. AAAI Press, 2020.

[23] Rodrigo Toro Icarte. *Reward Machines.* PhD thesis, University of Toronto, Canada, 2022.

[24] Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Anthony Valenzano, and Sheila A. McIlraith. Using reward machines for high-level task specification and decomposition in reinforcement learning. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 2112–2121. PMLR, 2018.

[25] Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Anthony Valenzano, and Sheila A. McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning. *J. Artif. Intell. Res.*, 73:173–208, 2022.

[26] Hado van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-Learning. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI*

Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA, pages 2094–2100. AAAI Press, 2016.

[27] Yin-Hao Wang, Tzuu-Hseng S. Li, and Chih-Jui Lin. Backward Q-learning: The combination of Sarsa algorithm and Q-learning. *Eng. Appl. Artif. Intell.*, 26(9):2184–2193, 2013.

[28] Christopher J. C. H. Watkins and Peter Dayan. Technical Note Q-Learning. *Mach. Learn.*, 8:279–292, 1992.

[29] Ronald J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Mach. Learn.*, 8:229–256, 1992.