Investigating Two Policy Gradient Methods Under Different Time Discretizations

by

Homayoon Farrahi

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

Continuous-time reinforcement learning tasks commonly use discrete time steps of fixed cycle times for actions. Choosing a small action-cycle time in such tasks allows reinforcement learning agents fast reaction and a more temporally detailed perception of the environment. The learning performance of both policy gradient and action-value methods, however, may deteriorate as the cycle time duration is reduced, which necessitates the tuning of the cycle time as a hyper-parameter. Since tuning an additional hyper-parameter is time-consuming, specifically for real-world robots, existing algorithms can benefit from having hyper-parameters that are robust to the choice of cycle time. In this thesis, we aim to study how changing the action-cycle time affects the performance of two prominent policy gradient algorithms PPO and SAC and investigate the efficacy of their widely-used hyper-parameter values across different cycle times. We explore how changing some of these hyper-parameters based on the cycle time can help or hinder the performance of these algorithms and inquire into and understand the relationship between them. These relationships are put forward as new hyper-parameters that can be adjusted based on the cycle time, and their effectiveness is examined and validated on simulated and real-world robotic tasks. We show that the new hyper-parameters, unlike the existing ones, can be more robust to different environments and cycle times and can enable hyper-parameter values tuned to a cycle time on a specific problem to be transferred to a different cycle time.

Preface

Results from this thesis involving the PPO experiments up to and including Chapter 6 were presented at the 3rd Robot Learning Workshop at the NeurIPS 2020 conference. The same material was included in a submission to and accepted at the ICRA 2021 conference, although we later withdrew the paper since, by the time of acceptance, we had significantly changed our hypotheses. Parts of this thesis including results from all of the chapters were submitted to and rejected from the UAI 2021 conference. The same material was submitted to and is under review at the CoRL 2021 conference. All of the above submissions were coauthored with my supervisor Prof. Rupam Mahmood.

To my parents

Acknowledgements

I am eternally grateful to my supervisor Prof. Rupam Mahmood for his invaluable guidance and regard to training rigorous scientists. He patiently explains the underlying reason for everything and encourages perseverance and long-term thinking, all of which I greatly treasure. I appreciate Prof. Richard Sutton and Prof. Michael Bowling for their thorough examination of this thesis.

I thank the Reinforcement Learning and Artificial Intelligence (RLAI) Lab, Alberta Machine Intelligence Institute (Amii), and the Canada CIFAR AI Chairs Program for funding this research. I am thankful to Kindred Inc. for their generous donation of the UR5 robotic arm and all of the amazing people in our Robot Lair group for the discussions.

Last but not least, I extend my gratitude to my mother Sara Vaziri, my father Farrokh Farrahi, and my sister Shiva Farrahi, whose unwavering love and support made my journey less demanding.

Contents

1 Introduction **1**

1.1 Reinforcement Learning With Small Action-Cycle Times . . . 2

1.2 Action-Cycle Time in the Real World 3

1.3 Adjusting Hyper-Parameters Based on the Action-Cycle Time 4

1.4 Related Works 5

1.5 Contributions 6

2 The Problem Setup **8**

2.1 The Markov Decision Process and the Objective 8

2.2 Time Limits in Reinforcement Learning 10

2.3 The Agent-Environment Interaction Loop 12

2.4 The Experiment Setup for Different Action-Cycle Times . . . 13

3 Policy Gradient Methods **17**

3.1 Likelihood-Ratio Policy Gradient Methods 17

3.2 The Proximal Policy Optimization (PPO) Algorithm 21

3.3 Reparameterization Policy Gradient Methods 24

3.4 The Soft Actor-Critic (SAC) Algorithm 25

4 Investigating the Baseline PPO Hyper-Parameters at Different Cycle Times **30**

4.1 The Reacher Task 30

4.2 Experiments 32

4.3 Results and Discussion 34

5 Setting Hyper-Parameters of PPO as a Function of the Action-Cycle Time 37

5.1 Our Proposed δt-Aware Hyper-Parameters of PPO 37

5.2 Experiments 39

5.3 Results and Discussion 40

6 Validating the δt-Aware Hyper-Parameters of PPO 47

6.1 The Double Pendulum Task 47

6.2 Validation on Double Pendulum 48

6.3 The Real-Robot Reacher Task 51

6.4 Validation on Real-Robot Reacher 52

7 Investigating the Discount Factor of SAC at Different Cycle Times 55

7.1 Experiments 55

7.2 Results and Discussion 56

7.3 Scaling Baseline γ as a Function of δt 58

8 Our Proposed δt-Aware Discount Factor of SAC 62

8.1 The δt-Aware Discount Factor 62

8.2 Experiments 63

8.3 Results and Discussion 63

9 Validating the δt-Aware Discount Factor of SAC 66

9.1 Validation on Double Pendulum 66

9.2 Validation on Real-Robot Reacher 67

10 Conclusion 73

References 75

List of Tables

4.1 Baseline Hyper-Parameters of PPO on the Reacher and the Double Pendulum Tasks 34
4.2 Tuned Hyper-Parameters of PPO on the Reacher Task 35

5.1 Baseline and δt-Aware Hyper-Parameters of PPO on the Reacher and the Double Pendulum Tasks 39

6.1 Hyper-Parameters of PPO on the Real-Robot Reacher Task . 53

7.1 Tuned γ of SAC on the Reacher Task 57

List of Figures

4.1 The Reacher Task with the robot arm, fingertip, and target . . 31

4.2 Learning curves for different δts with the baseline PPO hyper-parameters 34

4.3 Overall average return vs. δt for PPO with different hyper-parameter configurations 35

5.1 Learning curves of PPO using the δt-aware hyper-parameters . 41

5.2 Overall average return of PPO using the δt-aware hyper-parameters 42

5.3 Learning curves of PPO comparing the δt-aware hyper-parameters with ones where γ and λ are always exponentiated 43

5.4 Overall average return of PPO comparing the δt-aware hyper-parameters with exponentiated γ and λ 44

5.5 Learning curves of PPO comparing the δt-aware hyper-parameters with ones where γ and λ are kept constant 45

5.6 Overall average return of PPO comparing the δt-aware hyper-parameters with constant γ and λ 46

6.1 Learning curves of PPO using the baseline hyper-parameters on the Double Pendulum Task 49

6.2 Learning curves of PPO comparing the baseline and δt-aware hyper-parameters on the Double Pendulum Task 50

6.3 Overall average return of PPO for different δts on the Double Pendulum Task 50

6.4 The Real-Robot Reacher Task with the robot arm, the fingertip, and the target 51

6.5 Learning curves of PPO on the Real-Robot Reacher Task using hyper-parameters of Table 6.1 54
6.6 Learning curves of PPO on the Real-Robot Reacher Task using hyper-parameters of Table 5.1 54

7.1 Learning curves for different δts with the baseline SAC hyper-parameters 57
7.2 Overall average return vs. δt for SAC with different choices of γ 58
7.3 Learning curves of SAC for a sweep of γ values 59
7.4 Overall average return vs. a sweep of γ values for SAC 60

8.1 Learning curves of SAC comparing the baseline with the tuned and scaled γ 64
8.2 Learning curves of SAC comparing the baseline with the tuned γ 65
8.3 Overall average return of the two δt-aware hypotheses compared with the baseline γ for SAC 65

9.1 Learning curves of SAC using the baseline γ on the Double Pendulum Task 68
9.2 Learning curves of SAC comparing the baseline with the tuned and scaled γ on the Double Pendulum Task 68
9.3 Learning curves of SAC comparing the baseline with the tuned γ on the Double Pendulum Task 69
9.4 Overall average return of the two δt-aware hypotheses for γ of SAC compared with the baseline γ on the Double Pendulum Task 69
9.5 Learning curves of SAC on the Real-Robot Reacher Task for the sweep of γ values at $\delta t_0 = 40$ ms 71
9.6 Learning curves of SAC on the Real-Robot Reacher Task at $\delta t = 120$ ms 72
9.7 Overall average return of SAC comparing the baseline γ with our two δt-aware hypotheses on the Real-Robot Reacher Task 72

List of Algorithms

2.1 Agent-environment interaction loop 13
2.2 Agent-environment interaction loop for different δts 14

3.1 One-step actor-critic Initialize and Learn functions 20
3.2 PPO Initialize and Learn functions 22
3.3 SAC Initialize and Learn functions 29

Chapter 1

Introduction

Reinforcement learning algorithms have made remarkable strides in solving complex problems and have shown exceptional ability in learning complicated behaviors that are hard to design with conventional engineering approaches. These advances have been demonstrated in part on a wide range of continuous-time control tasks.

The continuous time in these tasks is usually discretized into time steps of equal duration called the *action-cycle time* δt, which refers to the time elapsed in the environment between two consecutive actions of an agent. This cycle time has been mostly considered as part of the environment, and the effects of its variation on the performance of popular recent algorithms have not been rigorously studied. Smaller δts have proved to be detrimental to the performance of different reinforcement learning (RL) algorithms, although they can potentially make RL agents more agile in action and keener in observation. The cycle time thus needs to be tuned and might have different optimal values for different tasks and environments. Tuning δt combined with other algorithm hyper-parameters can be costly and time-consuming on real-world robots due to the inherent real-time experience collection.

In this thesis, we explore the effects of changing the cycle time on the performance of two RL algorithms and provide recommendations for adapting their hyper-parameters based on the cycle time to make them more robust. We validate these recommendations on simulated and real-world robotic tasks.

1.1 Reinforcement Learning With Small Action-Cycle Times

Real-world robotic control tasks can benefit considerably from small cycle times, allowing fast interaction between the agent and the environment. Choosing a smaller δt entails taking actions more frequently and having faster reactions to changes in the environment, which, with a proper learning algorithm, may potentially improve the performance of reinforcement learning agents in many tasks and might even be crucial for others.

Even if a task does not require fast reactions, an agent may still benefit from a small δt by observing the environment more frequently, allowing it to observe important changes it might have otherwise missed with a large δt. The increased number of interactions with the environment, if paired with a capable algorithm, may also speed up learning by providing the agent the opportunity to elicit more information about the environment in less time.

Unfortunately, learning with existing action-value and policy gradient methods may be hindered by using a small cycle time. Previous works have brought to light the inadequacy of some RL methods as δt gets smaller and proposed new algorithms that are more suited to smaller δts and continuous time. For example, Baird (1994) illustrated that values of different actions in the same state at small δts get closer to each other, making learning the action-value function more sensitive to noise and function approximation error. Baird (1994) noted the collapse of the action-value function to the state value function in continuous time and provided a new algorithm, Advantage Updating, that learned the advantage function and the value function instead. Their approach was shown to be effective when applied to a continuous-time differential game by Harmon et al. (1995), and was later extended to deep Q-learning methods by Tallec et al. (2019), who provided recommendations for adjusting the algorithm parameters based on δt and showed the robustness of their algorithm—Deep Advantage Updating—to different δts.

Policy gradient methods are also susceptible to degraded performance as δt gets smaller. The variance of likelihood-ratio policy gradient estimates can explode as δt goes toward 0 as shown in an example by Munos (2006). They formulated a model-based policy gradient estimate using the gradient of the state with respect to the policy parameters assuming, however, that the system dynamics and the gradient of the reward function with respect to the state is known to the agent. These assumptions make their approach hard to apply to many practical tasks.

Although a small δt can technically provide benefits, in practice, it is difficult to learn with small cycle times as the aforementioned challenges inhibit the ability of RL algorithms to learn at small δts.

1.2 Action-Cycle Time in the Real World

Finding a cycle time that provides the right balance between task performance and learnability entails a search over several values of δt, which is time-consuming and costly for real-world robots, especially when combined with other hyper-parameters of learning algorithms. If learnability was not an issue, δt could be set to slightly more than the time it takes for an agent to calculate an action, which for deep RL methods involves performing a forward pass through a neural network.

Mahmood et al. (2018a) investigated the effect of using different values of δt on the learning performance of a 2-dimensional reaching task on a real-world robotic arm, and showed a δt in the middle of their chosen range to perform better than the smallest or the largest ones. Dulac-Arnold et al. (2020) showed that increasing δt hurts task performance when using fixed hyper-parameters. The same can be true when reducing δt with fixed hyper-parameter values as we will show later, corroborating the finding by Mahmood et al. (2018a) that a middle value of δt can perform better than the smallest or the largest ones.

The optimal cycle time might be different for different algorithms, tasks and robots, and the smallest δt afforded by the hardware is seldom the best, necessitating the tuning of δt along with other algorithm hyper-parameters. Although the cost of tuning one additional δt hyper-parameter can be manageable in simulated environments, real-world robots can only collect experience in real-time, which makes the tuning time-consuming, potentially more costly due to wear and tear, and thus impractical.

This work aims to study the effect of changing δt and other hyper-parameters on the performance of two famed algorithms. The algorithms are Proximal Policy Optimization (PPO), a popular on-policy policy gradient algorithm based on likelihood-ratio gradient estimation (Schulman et al. 2017), and Soft Actor Critic (SAC), a flourishing off-policy actor critic algorithm using the gradient of a reparameterized action-value estimate (Haarnoja et al. 2018a).

Although these algorithms have been applied successfully to real-world robotic tasks, the robustness of their performance and their hyper-parameters to different δts has been mostly overlooked. For instance, PPO has been used to solve the rubik's cube on a robotic hand (Akkaya et al. 2019), and PPO and SAC have been applied to a locomotion task on a quadruped robot (Tan et al. 2018, Haarnoja et al. 2018c). In these works, the action-cycle time has been kept fixed for all experiments.

1.3 Adjusting Hyper-Parameters Based on the Action-Cycle Time

Understanding how the cycle time affects learning and knowing its relationship to other hyper-parameters may lead to hyper-parameters that are more robust to different δts. For instance, using a small δt means that a sample batch of the same size will be collected in less time, possibly containing less information, and that future rewards will be discounted more heavily for the same discount factor and duration in real-time (Doya 2000). Having a set of guidelines and heuristics for adjusting the values of different algorithm hyper-parameters upon changing δt can enable the transfer of hyper-parameter values already tuned to one δt to a different δt on the same problem. Such guidelines may also potentially help reduce tuning cost, and lead to better task performance.

In this thesis, we demonstrate the ineffectiveness of the *baseline* hyper-parameter values of PPO and SAC when learning at δts other than the default of the simulated environment. These baseline values are provided by the original works introducing the algorithms, which are tuned for a set of tasks with default δts and widely-used in many other works (e.g., Ramstedt & Pal 2019, Fujimoto et al. 2018). We also refer to any such hyper-parameter values tuned to a particular δt as baseline values in this work. We then propose a set of modifications, based on δt, to these hyper-parameters to improve upon the performance of the baseline values. Our results highlight the lack of robustness of the baseline hyper-parameter values to different environments in both algorithms. We validate our proposed set of modifications on a simulated and a real-world robotic task.

1.4 Related Works

There are other works that are related to the action-cycle time and continuous-time reinforcement learning. The continuous-time version of the Bellman equation, Hamilton-Jacobi-Bellman (HJB) equation, has been used to derive RL algorithms for the continuous-time case (Kim & Yang 2020). Munos and Bourgine (1998) derived a value function update rule from the HJB equation and proposed a model-based RL algorithm. Doya (1996) derived the continuous-time TD error and used it to develop methods for learning the value function and to extend the actor-critic method to the continuous-time case. They extended this work in Doya (2000) and showed the robustness of their methods across different cycle times and simulated environments. Lee and Sutton (2021) developed the fundamental theory for applying policy iteration methods to continuous-time systems and provided case studies in discounted RL and optimal control contexts.

The time it takes for an agent to output an action after an observation—the action delay—has been investigated in simulated environments (Firoiu et al. 2018, Chen et al. 2021) and is of particular importance in real-world robotics applications since, unlike simulated environments, real-world environments do not halt their progress as the agent calculates an action (Mahmood et al. 2018a, Chen et al. 2021).

Travnik et al. (2018) showed the adverse effects of large action delays on task performance and minimized this delay by reordering algorithmic steps. Dulac-Arnold et al. (2020) observed deteriorating task performance with increasing action and observation delay. Ramstedt and Pal (2019) introduced the Real-Time Markov Decision Process and Real-Time Actor-Critic. In their framework, however, δt should ideally be set equal to the time for a forward pass of the policy, which could vary greatly for different policy architectures. In this work, we focus on the cycle time issues and assume that the chosen δts always fit the forward pass for action calculations.

Reinforcement learning in continuous-time has been further studied in the context of optimal control of dynamical systems (Vamvoudakis & Lewis 2010, Bhasin et al. 2013, Modares & Lewis 2014), game theory (Johnson et al. 2011, Li et al. 2014), and deep neural networks (Zambrano et al. 2015, Xiao et al. 2020, Du et al. 2020). Deep reinforcement learning methods have made exceptional strides in simulated environments (Mnih et al. 2015, Berner et al. 2019) and have been successfully applied to real-world robotic tasks (Tan et al. 2018, Haarnoja et al. 2018c, Akkaya et al. 2019). However, the robustness of these methods and their hyper-parameters to different cycle times has been mostly ignored, which is the subject of our focus in this study.

1.5 Contributions

The contributions of this work can be summarized in four main categories:

- We demonstrate that the baseline hyper-parameter values of PPO and SAC are not robust to different δts. We show that the performance of these algorithms changes significantly across different δts when other hyper-parameters are kept constant at their baseline values. This is especially problematic for real-world robotic tasks, as it entails tuning all algorithm hyper-parameters every time a new δt is experimented with on a given task for which the baseline values performed well.

- We propose novel approaches for adapting the hyper-parameters of PPO and SAC based on δt and show that our proposed modifications make these hyper-parameters more robust to different δts than the baseline values on the original simulated task.

- We validate our recommendations on *held-out* simulated and real-world robotic tasks that were not used in the investigations for devising the recommendations. These new hyper-parameters can perform better than the baseline values when δt changes on both tasks. The improved robustness of these hyper-parameters to different δts is important in the real-world task, where experience collection is expensive, as it allows for hyper-parameter values that have been tuned to a particular δt to be transferred to a different δt on the same task.

- We conduct these experiments on existing tasks modified carefully to accommodate and compare fairly across different δts, which can be used for benchmarking the impact of δt on new algorithms in the future. We open-source our implementation of these tasks and experiments to facilitate further studies on the action-cycle time. A video of the experiments on the real-world robot can be seen at `https://www.youtube.com/watch?v=tmo5fWGRPtk`. The code for all experiments is available at `https://github.com/homayoonfarrahi/cycle-time-study`.

Chapter 2

The Problem Setup

This chapter presents the problem setup and its implementation that we use for all of our experiments. We investigate the robustness of algorithm hyper-parameters to different time discretizations by formulating the problem as an undiscounted episodic Markov decision process (MDP) with time limits. We describe the discounted MDP since it is a generalization of the undiscounted MDP, and the discounted MDP is the setup that our solution methods use. We note that our undiscounted problem formulation can be retrieved from the discounted MDP by setting the discount factor equal to one. Next, the issue of time limits in reinforcement learning tasks and potential solutions to it are explained. We then discuss how the agent-environment interaction is implemented and how it is modified to support different action-cycle time δts.

2.1 The Markov Decision Process and the Objective

We use the undiscounted episodic reinforcement learning framework as described in Sutton and Barto (2018). Although our problem setup is undiscounted, we describe it for the discounted MDP and set the discount factor equal to one. The problem is formulated as a Markov decision process in which the agent and the environment interact at discrete non-negative integer time steps t starting from 0. All possible non-terminal and terminal states that the agent receives are represented by the sets $\mathcal{S}$ and Ω respectively with $\mathcal{S} \cap \Omega = \emptyset$. All possible actions that the agent can take are represented by the set $\mathcal{A}$.

The agent interacts with the environment through a sequence of episodes by starting a new episode after the previous one is terminated. Each episode comprises a series of interactions from the starting time step $t = 0$ to the terminal time step $t = T$ and begins with the start state $S_0 \in \mathcal{S}$ sampled from the start state density $d_0(s)$.

In the following time steps t, the agent receives the current state of the environment $S_t \in \mathcal{S}$ and uses a probability density π, called a policy, to select an action $A_t \sim \pi(\cdot|S_t)$ with $A_t \in \mathcal{A}$ and apply it to the environment. In the next time step $t+1$, the environment proceeds to the next state $S_{t+1} \in \mathcal{S} \cup \Omega$ and generates a scalar reward signal $R_{t+1} \in \mathbb{R}$ based on the probability density $S_{t+1}, R_{t+1} \sim p(\cdot, \cdot|S_t, A_t)$, which defines the dynamics of the environment. The newly generated next state and reward are given to the agent, and the cycle repeats until a terminal reward R_T is received and a terminal state $S_T \in \Omega$ is reached at the terminal time step T. The terminal time step T is a random variable and can vary from one episode to another. The sequence below shows the order of states, actions, and rewards with their time steps for a single episode from start to finish:

$$S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_t, A_t, R_{t+1}, S_{t+1}, \ldots, S_{T-1}, A_{T-1}, R_T, S_T.$$

In this framework, the return for time step t is denoted by G_t and is defined as the discounted sum of all future rewards until the end of the episode

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$$

with $\gamma \in [0, 1]$ being the discount factor that is used to discount the value of future rewards to obtain their present value. The value of a state s under policy π is the expected return from that state with the policy π followed from that state onward

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t | S_t = s\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \,\middle|\, S_t = s \right]. \tag{2.1}$$

When the policy π is parameterized by $\boldsymbol{\theta}$, the goal of the agent is to change the parameters $\boldsymbol{\theta}$ to maximize the expected value of the initial state, which is called the performance measure $J^{\gamma}(\boldsymbol{\theta})$ and defined as

$$J^{\gamma}(\boldsymbol{\theta}) \doteq \int_{s} d_0(s)\ v_{\pi_{\boldsymbol{\theta}}}(s). \tag{2.2}$$

In our problem formulation, the discount factor $\gamma = 1$, and we seek to maximize the expected undiscounted episodic return $J^{1}(\boldsymbol{\theta})$.

2.2 Time Limits in Reinforcement Learning

Many reinforcement learning tasks have a fixed maximum number of time steps, called a *time limit*, upon reaching which the current episode ends and the next one starts by resetting the environment according to the start state distribution. Time limits are often used to impose episodes on tasks that are truly continuing in nature. Time limits together with robust resets can also help recover agents from a lengthy unproductive trajectory and diversify the learning experience. Although both simulated and real-world robotic tasks that we use for our investigations have time limits, we did not address the potential issues that they may cause with reinforcement learning algorithms. For completeness, we describe these issues and potential solutions to them in this section.

Some reinforcement learning tasks are sound without using time limits, but can lead to faster learning with them included. For instance, in the original Mountain Car task (Singh & Sutton 1996), stochastic policies can eventually reach the goal and terminate the episode without using time limits. Nonetheless, time limits of 10,000 steps are used to expedite learning. In this particular task, as an agent learns to reach the goal efficiently, episodes are less often terminated from reaching the time limit. On the other hand, for a pole-balancing task where the goal of the agent is to keep a pole upright on a moving cart, episodes can get longer and longer as the agent learns. In this scenario, time limits can be used to end the episodes and are encountered more often as the learning progresses.

When a time limit is imposed on a task, should the modified task be considered an episodic or a continuing one? If it is considered an episodic task, the state at the time limit should be treated as a terminal state. Moreover, the optimal action in the same environmental state might be different depending on the amount of time remaining in the episode. However, for the original environment without the time limit, this information about time was not included in the state. This means that the states of the original environment will no longer have the Markov property if used for the modified task with time limits. Although effective learned behavior has been demonstrated on many tasks with time limits, the non-Markovian state in these tasks may prevent agents from learning the optimal policy. For example, in a time-limited task where a one-legged robot has to learn to run forward as far as possible, using the non-Markovian states the agent can learn to run forward up until the end of the episode. Yet, if the agent has access to the time information, it can learn the optimal policy, which involves diving forward and possibly crash landing near the time limit to travel further and obtain a higher return.

If a task with time limits is considered episodic, the agent should be able to predict the arrival of the terminal state. This can be achieved by including the history of past observations in the state. Alternatively, the state can be augmented to include information about the passage of time in the episode by including the time step in the state in the simplest case. An agent that has access to this knowledge can learn to favor actions that lead to more short-term rewards as it approaches the time limit. Without the time information, the agent cannot discern between similar states that occur closer to or further from the time limit and has to learn to compromise.

On the other hand, if time-limited tasks are considered continuing, the states that the agent is in upon reaching the time limit should not be treated as terminal states. That is, the agent should learn to behave as if the episode would have continued beyond the time limit. In addition, it is unclear how the discount factor of a continuing problem can be chosen and how the performance should be measured on such a task. With time limits, typically the undiscounted truncated return is reported as the performance measure.

For the states reached at the time limit not to be treated as terminal states, one can bootstrap from them using their state values. This is in contrast to terminal states, which are not bootstrapped from since their state values are considered zero. It is not clear what solution algorithms using this technique arrive at, as it bootstraps from states that the agent does not visit in the next time step. Additionally, the issues of the discount factor and the performance measure remain unaddressed for continuing problems, which can perhaps be tackled using the average reward setting of reinforcement learning. Some of the above issues and potential solutions were discussed by Pardo et al. (2018).

2.3 The Agent-Environment Interaction Loop

In this section, we describe the agent-environment interaction algorithmically by separating the specific details of the learning algorithms from their common parts. This separation allows us to have a common agent-environment interaction loop for the two algorithms that we experiment on, and to show later in this chapter how it is modified to support different cycle times without changing the details of the specific algorithm. Here, we describe the common part, which outlines the interaction loop between the agent and the environment and when the learning updates happen. The second part is described in the next chapter and explains the specific learning update of an algorithm in more detail.

The parameter Ψ represents the set of all parameters and hyper-parameters for a specific algorithm. Its constituents are defined by the *Initialize* function that is defined separately for each algorithm. This encapsulation simplifies the discription of this general agent-environment interaction loop since algorithm-specific parameters are defined where they are used.

The interaction loop uses a buffer to store the experience of the agent. At each time step, the most recent interaction between the agent and the environment is stored as an experience sample $(S_i, A_i, R_{i+1}, S'_{i+1}, T_{i+1})$ with $T_{i+1} \doteq 1$ if S'_{i+1} is a terminal state and 0 otherwise. If the buffer has reached its full capacity of b samples, the new interaction will overwrite the oldest one in the buffer.

Algorithm 2.1 lists the agent-environment interaction loop. All parameters

Algorithm 2.1: Agent-environment interaction loop

```
$\Psi \doteq$ Initialize()
Retrieve from $\Psi$: learning period $U$, batch size $b$, parameterized policy
  $\pi_{\boldsymbol{\theta}}(a|s)$
Initialize Buffer $B$ with capacity $b$
Initialize $S_0 \sim d_0(\cdot)$
for environment step $i = 0, 1, 2, ...$ do
    Calculate action $A_i \sim \pi_{\boldsymbol{\theta}}(\cdot|S_i)$
    Apply $A_i$ and observe $R_{i+1}, S'_{i+1}$
    $T_{i+1} \doteq \mathbb{1}_{S'_{i+1} \text{ is terminal}}$
    Store transaction in buffer $B_i = \big(S_i, A_i, R_{i+1}, S'_{i+1}, T_{i+1}\big)$
    if $i + 1 \bmod U = 0$ then
        $\Psi \doteq$ Learn$(B, \Psi)$
    if $T_{i+1} = 1$ then
        Sample $S_{i+1} \sim d_0(\cdot)$
    else
        $S_{i+1} \doteq S'_{i+1}$
end
```

are first initialized by the algorithm-specific *Initialize* function. The agent repeatedly selects and applies an action based on the current state, observes the reward and the next state, and stores the transaction in the buffer. If the agent is at the end of a learning period U, it calls the algorithm-specific *Learn* function to update its parameters.

2.4 The Experiment Setup for Different Action-Cycle Times

We set up our experiments in a manner that allows the agent and the environment to interact with different cycle time δts. There are at least two approaches for achieving this problem setup. The first is to use the common agent-environment interaction loop of Algorithm 2.1 and change the δt of an environment between different runs. In this case, the rewards of the environment need to be scaled by δt to ensure that the returns are comparable between different δts. If we take this approach, setting the δt of a simulated environment will entail changing the time interval of the underlying physics engine, which may cause inconsistencies between different δts. For instance, if

Algorithm 2.2: Agent-environment interaction loop for different δts

```
$\Psi \doteq$ Initialize()
Retrieve from $\Psi$: action-cycle time $\delta t$, environment time step $\delta t_{\text{env}}$,
  learning period $U$, batch size $b$, parameterized policy $\pi_{\boldsymbol{\theta}}(a|s)$
Initialize Buffer $B$ with capacity $b$
Initialize $S_0 \sim d_0(\cdot)$
$j \doteq 0$ // episode step
$k \doteq 0$ // agent step
for environment step $i = 0, 1, 2, ...$ do
    if $j$ mod $\delta t / \delta t_{\text{env}} = 0$ then
        $\tilde{S}_k \doteq S_i$
        $\tilde{R}_{k+1} \doteq 0$
        Calculate action $\tilde{A}_k \sim \pi_{\boldsymbol{\theta}}(\cdot|S_i)$
    Apply $\tilde{A}_k$ and observe $R_{i+1}, S'_{i+1}$
    $\tilde{R}_{k+1} \doteq \tilde{R}_{k+1} + R_{i+1}$
    $T_{i+1} \doteq \mathbb{1}_{S'_{i+1} \text{ is terminal}}$
    if $j + 1$ mod $\delta t / \delta t_{\text{env}} = 0$ or $T_{i+1} = 1$ then
        Store transaction in buffer $B_k = \left(\tilde{S}_k, \tilde{A}_k, \tilde{R}_{k+1}, S'_{i+1}, T_{i+1}\right)$
        if $k + 1$ mod $U = 0$ then
            $\Psi \doteq$ Learn$(B, \Psi)$
        $k \doteq k + 1$
    $j \doteq j + 1$
    if $T_{i+1} = 1$ then
        Sample $S_{i+1} \sim d_0(\cdot)$
        $j \doteq 0$
    else
        $S_{i+1} \doteq S'_{i+1}$
end
```

the physics engine takes a step of length $\delta t = 16$ ms, the resulting transition might be different from taking two steps of lengths $\delta t = 8$ ms.

A second approach for setting up experiments with different δts, and the one that we use for all of our experiments, is to run the environment at a fixed environment time step of δt_{env} and simulate other δts as integer multiples of δt_{env}. We implement this setup by modifying the common agent-environment interaction to run the environment at δt_{env}, and making the agent interact with the environment every multiple environment steps. For this implementation, the rewards of the environment do not need to be scaled by δt, but are accumulated between two consecutive agent interactions. Compared to changing the environment time step for each δt, our approach of keeping it constant is beneficial since it dispenses with the inconsistencies that can arise from running the physics engine at different time intervals. The environment changes in the same way if two agents with two different δts provide the same sequence of actions. Algorithm 2.2 lists the common agent-environment interaction loop that is modified to support experiments with different δts. This experiment setup is not specific to a particular learning algorithm, which can be plugged into Algorithm 2.2 by defining the *Initialize* and *Learn* functions.

We use three different indices to implement this: i, j, and k. The i index behaves similarly to the original algorithm and is incremented every environment time step of δt_{env}. The j index is incremented with the same frequency as i, but is reset to 0 at the start of each episode. Its purpose is to ensure a new episode always begins with a new action. The k index is incremented at the lower frequency of our desired cycle time δt with which the agent operates. In addition, the $\tilde{S}$, $\tilde{A}$, and $\tilde{R}$ variables are always indexed using k and respectively represent the state, action, and the reward at the lower frequency cycle time of δt.

Once every δt milliseconds, a new action $\tilde{A}_k$ is selected based on the current state $\tilde{S}_k \doteq S_i$ and is repeatedly applied to the environment until the next interaction. The action is repeated because in our environments the actions set the speeds or torques of joints, values of which remain fixed until the next action is applied. The variable $\tilde{R}_{k+1}$ accumulates all of the observed rewards R_{i+1} for this period without discounting since our problem setup is undiscounted. At the end of the period, or if the environment reaches a terminal state, the experience sample $\left(\tilde{S}_k, \tilde{A}_k, \tilde{R}_{k+1}, S'_{i+1}, T_{i+1}\right)$ is stored in the buffer with $\tilde{S}_k$ and S'_{i+1} as the current and the next state of the sample respectively. A learning update is then made in the same way as the original algorithm if it is time to do so.

We defined our problem setup as a Markov decision process in which the goal is to maximize the expected undiscounted episodic return. We discussed how time limits can affect and should be treated in episodic and continuing reinforcement learning tasks differently, although our tasks were not modified to take time limits into account. We then presented the agent-environment interaction loop that is common between the two algorithms that we study and modified it to support running experiments with different δts. Next chapter describes the solution methods.

Chapter 3

Policy Gradient Methods

We describe the algorithms that we experiment with and their implementations in this chapter. We discuss the likelihood-ratio and reparameterization policy gradient theorems and show how the Proximal Policy Optimization (PPO) and the Soft Actor-Critic (SAC) algorithms can be derived from them. We describe the implementation of these algorithms by specifying their *Initialize* and *Learn* functions that can be plugged into the general agent-environment interaction loop from Section 2.4.

3.1 Likelihood-Ratio Policy Gradient Methods

Instead of defining a policy based on directly maximizing value and action-value estimates, policy gradient methods use parameterized distributions to represent the probability of taking an action a in a state s. As discussed by Sutton and Barto (2018), policy gradient methods have three advantages over the former value-based methods: the policy might be a simpler function to approximate, can simultaneously be optimal and allow selection of actions with arbitrary probabilities, and can become deterministic in the limit.

Although our problem formulation is undiscounted ($\gamma = 1$), our solution methods use discounting with $\gamma \in [0, 1]$. We defined the value function in (2.1). The action-value function $q_{\pi_{\boldsymbol{\theta}}}(s, a)$ represents the expected discounted episodic return when the agent is in state s, takes action a, and thereafter follows policy $\pi_{\boldsymbol{\theta}}$:

$$q_{\pi_{\boldsymbol{\theta}}}(s,a) \doteq \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[\sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right].$$

The value function at state s can be written in terms of the action-value function. It is equal to the average of different action values in state s weighted by the probability of taking each action:

$$\upsilon_{\pi_{\boldsymbol{\theta}}} = \int_a \pi_{\boldsymbol{\theta}}(a|s) q_{\pi_{\boldsymbol{\theta}}}(s,a)\ da.$$

The policy $\pi_{\boldsymbol{\theta}}(a|s)$ is optimized by changing its parameters $\boldsymbol{\theta}$ to maximize the performance measure $J^\gamma(\boldsymbol{\theta})$ defined in (2.2). The performance measure can be written in terms of the action-value function as

$$\begin{aligned} J^\gamma(\boldsymbol{\theta}) &\doteq \int_s d_0(s) \upsilon_{\pi_{\boldsymbol{\theta}}}(s)\ ds \\ &= \int_s d_0(s) \int_a \pi_{\boldsymbol{\theta}}(a|s) q_{\pi_{\boldsymbol{\theta}}}(s,a)\ da\ ds. \end{aligned} \tag{3.1}$$

The performance measure $J^\gamma(\boldsymbol{\theta})$ is often optimized using gradient ascent. One approach to do so is to derive the gradient of $J^\gamma(\boldsymbol{\theta})$ with respect to the parameters $\boldsymbol{\theta}$ from (3.1):

$$\nabla J^\gamma(\boldsymbol{\theta}) = \int_s d_0(s) \int_a \Big(\nabla \pi_{\boldsymbol{\theta}}(a|s) q_{\pi_{\boldsymbol{\theta}}}(s,a) + \pi_{\boldsymbol{\theta}}(a|s) \nabla q_{\pi_{\boldsymbol{\theta}}}(s,a)\Big)\ da\ ds,$$

which after repeated expanding and unrolling of the term $\nabla q_{\pi_{\boldsymbol{\theta}}}(s,a)$ leads to the policy gradient theorem proven by Marbach and Tsitsiklis (2001) and independently by Sutton et al. (2000)

$$\begin{aligned} \nabla J^\gamma(\boldsymbol{\theta}) &\propto \int_s d^{\gamma,\pi_{\boldsymbol{\theta}}}(s) \int_a q_{\pi_{\boldsymbol{\theta}}}(s,a) \nabla \pi_{\boldsymbol{\theta}}(a|s)\ da\ ds, \\ d^{\gamma,\pi_{\boldsymbol{\theta}}}(s) &\doteq \int_x \sum_{k=0}^{\infty} \gamma^k d_0(x) p(x \to s, k, \pi_{\boldsymbol{\theta}})\ dx \end{aligned}$$

with $p(x \to s, k, \pi_{\boldsymbol{\theta}})$ representing the probability of going from state x to state s in k steps under policy $\pi_{\boldsymbol{\theta}}$, and $d^{\gamma,\pi_{\boldsymbol{\theta}}}(s)$ as the discounted on-policy state distribution, or the distribution with which states are visited when following policy $\pi_{\boldsymbol{\theta}}$, appropriately discounted by γ through time.

For an agent that follows the policy $\pi_{\boldsymbol{\theta}}(a|s)$, states are visited and actions are taken according to this policy, and hence the policy gradient theorem can be written in its equivalent expected form under policy $\pi_{\boldsymbol{\theta}}$ as

$$\begin{aligned}
\nabla J^{\gamma}(\boldsymbol{\theta}) &= \mathbb{E}_{d^{\pi_{\boldsymbol{\theta}}}}\left[\gamma^t \int_a \pi_{\boldsymbol{\theta}}(a|S_t) q_{\pi_{\boldsymbol{\theta}}}(S_t, a) \frac{\nabla \pi_{\boldsymbol{\theta}}(a|S_t)}{\pi_{\boldsymbol{\theta}}(a|S_t)}\ da\right] \\
&= \mathbb{E}_{d^{\pi_{\boldsymbol{\theta}}}, \pi_{\boldsymbol{\theta}}}\left[\gamma^t q_{\pi_{\boldsymbol{\theta}}}(S_t, A_t) \frac{\nabla \pi_{\boldsymbol{\theta}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}}(A_t|S_t)}\right] \\
&= \mathbb{E}_{d^{\pi_{\boldsymbol{\theta}}}, \pi_{\boldsymbol{\theta}}}\left[\gamma^t G_t \nabla \log \pi_{\boldsymbol{\theta}}(A_t|S_t)\right], \qquad (3.2) \\
d^{\pi_{\boldsymbol{\theta}}}(s) &\doteq \int_x \sum_{k=0}^{\infty} d_0(x) p(x \to s, k, \pi_{\boldsymbol{\theta}})\ dx,
\end{aligned}$$

where $G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$ is the return from time step t. Notice that the expectation now assumes that the states are visited according to the undiscounted on-policy state distribution $d^{\pi_{\theta}}$, and the discounting term γ^t has moved from $d^{\gamma, \pi_{\boldsymbol{\theta}}}$ into the expectation. The $\gamma^t G_t \nabla \log \pi_{\boldsymbol{\theta}}(A_t|S_t)$ term is referred to as the likelihood-ratio policy gradient estimate. It can be calculated for every time step and used to update the parameters $\boldsymbol{\theta}$ in the direction of the gradient to maximize $J^{\gamma}(\boldsymbol{\theta})$, which leads to the REINFORCE algorithm by Williams (1987, 1992). The sample gradient can also be arrived at by differentiating the following sample surrogate objective with respect to $\boldsymbol{\theta}$:

$$L_{\boldsymbol{\theta}} \doteq -\gamma^t G_t \log \pi_{\boldsymbol{\theta}}(A_t|S_t).$$

The variance of the above sample surrogate objective can be reduced by subtracting from G_t a baseline that does not depend on actions and therefore adds no bias. A common choice for this baseline is the parameterized value estimate $\hat{v}_{\mathbf{w}}(s)$, which can be learned in tandem with the policy by minimizing the semi-gradient TD(0) (Sutton & Barto 2018) objective $L_{\mathbf{w}}$ below. In addition, the return G_t in the policy objective can be replaced by the one-step return $G_{t:t+1} \doteq R_{t+1} + \gamma \hat{v}_{\mathbf{w}}(S'_{t+1})$, which has lower variance and results in

Algorithm 3.1: One-step actor-critic Initialize and Learn functions

```
Function Initialize():
    $\Psi \doteq \{$ learning period $U$, batch size $b$, discount factor $\gamma$,
     parameterized policy $\pi_{\boldsymbol{\theta}}(a|s)$, parameterized value estimate
     $\hat{v}_{\mathbf{w}}(s)$, current discount $I$, policy learning rate $\eta^{\boldsymbol{\theta}}$, value estimate
     learning rate $\eta^{\mathbf{w}}\}$
    Denote network parameters by $\Psi.\boldsymbol{\theta}$, $\Psi.\mathbf{w}$
    Initialize parameters $\Psi.\boldsymbol{\theta}$, $\Psi.\mathbf{w}$
    $I \doteq 1$
    return $\Psi$

Function Learn($B$, $\Psi$):
    Retrieve the only transaction in the buffer $S_t, A_t, R_{t+1}, S'_{t+1}, T_{t+1}$
    $\boldsymbol{\theta} \doteq \Psi.\boldsymbol{\theta}$ ; $\mathbf{w} \doteq \Psi.\mathbf{w}$
    $\overline{\mathbf{w}} \doteq \mathbf{w}$
    $\delta_t \doteq R_{t+1} + (1 - T_{t+1})\gamma\hat{v}_{\overline{\mathbf{w}}}(S'_{t+1}) - \hat{v}_{\mathbf{w}}(S_t)$
    $\boldsymbol{\theta} \doteq \boldsymbol{\theta} + \eta^{\boldsymbol{\theta}} I \delta_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t|S_t)$
    $\mathbf{w} \doteq \mathbf{w} + \eta^{\mathbf{w}} I \delta_t \nabla_{\mathbf{w}} \hat{v}_{\mathbf{w}}(S_t)$
    if $T_{i+1} = 1$ then $I \doteq 1$
    else $I \doteq \gamma I$
    $\Psi.\boldsymbol{\theta} \doteq \boldsymbol{\theta}$ ; $\Psi.\mathbf{w} \doteq \mathbf{w}$
    return $\Psi$
```

faster learning. The new objectives $L_{\boldsymbol{\theta}}$ and $L_{\mathbf{w}}$ can be written as

$$\begin{aligned}
L_{\boldsymbol{\theta}} &\doteq -\gamma^t \Big(G_{t:t+1} - \hat{v}_{\mathbf{w}}(S_t)\Big) \log \pi_{\boldsymbol{\theta}}(A_t|S_t) \\
&= -\gamma^t \Big(R_{t+1} + \gamma\hat{v}_{\mathbf{w}}(S'_{t+1}) - \hat{v}_{\mathbf{w}}(S_t)\Big) \log \pi_{\boldsymbol{\theta}}(A_t|S_t) \\
&= -\gamma^t \delta_t \log \pi_{\boldsymbol{\theta}}(A_t|S_t) \\
L_{\mathbf{w}} &\doteq \frac{1}{2}\gamma^t \Big(R_{t+1} + \gamma\hat{v}_{\overline{\mathbf{w}}}(S'_{t+1}) - \hat{v}_{\mathbf{w}}(S_t)\Big)^2,
\end{aligned}$$

and together they create a one-step actor-critic method. The symbol δ_t is known as the TD error. The value estimate parameters $\overline{\mathbf{w}}$ are equal to $\mathbf{w}$, but are denoted as such since they do not take part in the calculation of the gradient with respect to $\mathbf{w}$, hence the name semi-gradient. Taking the gradient of these objectives leads to the update rules below with learning rates $\eta^{\boldsymbol{\theta}} > 0$ and $\eta^{\mathbf{w}} > 0$.

$$\begin{aligned}
\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \eta^{\boldsymbol{\theta}} \gamma^t \delta_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t|S_t) \\
\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \eta^{\mathbf{w}} \gamma^t \delta_t \nabla_{\mathbf{w}} \hat{v}_{\mathbf{w}}(S_t)
\end{aligned}$$

Algorithm 3.1 uses these objectives and describes the *Initialize* and *Learn* functions of this actor-critic algorithm that can be plugged into the general agent-environment interaction loop in Algorithm 2.1. This algorithm performs an update at every time step using only the most recent experience sample, which means $b = U = 1$. The $(1 - T_{t+1})$ term ensures that the estimated value of terminal states is considered zero.

3.2 The Proximal Policy Optimization (PPO) Algorithm

The Proximal Policy Optimization algorithm was developed by Schulman et al. (2017) and significantly outperformed many other policy gradient methods at the time of its introduction. The simplicity and robustness of PPO has led to its widespread adoption in many real-world robotic tasks (Mahmood et al. 2018b, Tan et al. 2018, Akkaya et al. 2019). In our implementation of PPO, we wait until the episode is over to do the lengthy *Learn* operation.

In PPO, the batch size b is equal to the learning period U. This means that between each two consecutive learning updates, all experience samples in the buffer are replaced by b new ones, such that each learning update only uses the latest b agent-environment interactions. This buffer that is filled anew for each learning update is also referred to as a batch of data.

PPO uses the likelihood-ratio policy gradient estimate. Its objective can be derived from the gradient estimate of REINFORCE from (3.2) by replacing the return G_t with the λ-return $G_t^\lambda \doteq \lambda^{T-t-1} G_t + (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n}$ using the n-step return $G_{t:t+n} \doteq \gamma^n \hat{v}_{\overline{\mathbf{w}}}(S_{t+n}) + \sum_{j=1}^{n} \gamma^{j-1} R_{t+j}$, subtracting from it a value estimate baseline $\hat{v}_{\mathbf{w}}(s)$, and dropping the discounting term γ^t, although discounting is still used in the calculation of G_t^λ. The value estimate parameters $\overline{\mathbf{w}}$ are equal to $\mathbf{w}$, but do not contribute to the calculation of gradient with respect to $\mathbf{w}$. The dropping of the discount factor γ^t makes the gradient estimate biased and is prevalent in many policy gradient algorithms as discussed by Nota and Thomas (2020). They proved that this biased estimate is not the gradient of any objective. Nonetheless, we drop the γ^t term to be

Algorithm 3.2: PPO Initialize and Learn functions

```
Function Initialize():
    $\Psi \doteq \{$ learning period $U$, number of epochs $N$, batch size $b$,
      mini-batch size $m$, discount factor $\gamma$, trace-decay parameter $\lambda$,
      clipping parameter $\epsilon$, parameterized policy $\pi_{\boldsymbol{\theta}}(a|s)$, parameterized
      value estimate $\hat{v}_{\mathbf{w}}(s)$, learning rate $\eta$ $\}$
    Denote network parameters by $\Psi.\boldsymbol{\theta}$, $\Psi.\mathbf{w}$
    Initialize parameters $\Psi.\boldsymbol{\theta}$, $\Psi.\mathbf{w}$
    return $\Psi$
Function Learn($B$, $\Psi$):
    $\boldsymbol{\theta} \doteq \Psi.\boldsymbol{\theta}$ ; $\mathbf{w} \doteq \Psi.\mathbf{w}$
    $\overline{\mathbf{w}} \doteq \mathbf{w}$
    for $t = 0, 1, 2, ..., b$ do
        Retrieve transaction from buffer $S_t, A_t, R_{t+1}, S'_{t+1}, T_{t+1}$
        $T \doteq \min\{j \mid j \in \mathbb{N} \ \wedge \ j > t \ \wedge \ T_j = 1\}$
        $G_t^\lambda \doteq \lambda^{T-t-1} G_t + (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n}$,
          where $G_{t:t+n} \doteq \gamma^n \hat{v}_{\overline{\mathbf{w}}}(S_{t+n}) + \sum_{j=1}^{n} \gamma^{j-1} R_{t+j}$
        $\hat{h}_t \doteq G_t^\lambda - \hat{v}_{\mathbf{w}}(S_t)$ // advantage estimate
    end
    $\tilde{h} \doteq \text{normalize}(\hat{h})$
    $D \doteq \left((S_t, A_t, \tilde{h}_t, G_t^\lambda)\right)_{t=0}^{b}$
    $\boldsymbol{\theta}_{old} \doteq \boldsymbol{\theta}$
    for epoch $e = 1, 2, ..., N$ do
        $\tilde{D} \doteq \text{shuffle}(D)$
        Slice $\tilde{D}$ into $\lceil \frac{b}{m} \rceil$ mini-batches
        for each mini-batch $M$ do
            $\boldsymbol{\theta} \doteq \boldsymbol{\theta} - \eta \frac{1}{m} \sum_{(S_k, A_k, \tilde{h}_k) \in M} \nabla_{\boldsymbol{\theta}} L_{\boldsymbol{\theta},k}$
              using (3.3), where
              $L_{\boldsymbol{\theta},k} \doteq -\min\left(\rho_k(\boldsymbol{\theta})\tilde{h}_k, \rho_k^{\text{clip}}(\boldsymbol{\theta})\tilde{h}_k\right)$,
              $\rho_k(\boldsymbol{\theta}) \doteq \frac{\pi_{\boldsymbol{\theta}}(A_k|S_k)}{\pi_{\boldsymbol{\theta}_{old}}(A_k|S_k)}$, and
              $\rho_k^{\text{clip}}(\boldsymbol{\theta}) \doteq \text{clip}(\rho_k(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon)$
            $\mathbf{w} \doteq \mathbf{w} + \eta \frac{1}{m} \sum_{(S_k, G_k^\lambda) \in M} 2\left(G_k^\lambda - \hat{v}_{\mathbf{w}}(S_k)\right) \nabla_{\mathbf{w}} \hat{v}_{\mathbf{w}}(S_k)$
        end
    end
    $\Psi.\boldsymbol{\theta} \doteq \boldsymbol{\theta}$ ; $\Psi.\mathbf{w} \doteq \mathbf{w}$
    return $\Psi$
```

consistent with the original implementation of PPO.

Since the policy parameters $\boldsymbol{\theta}$ are updated multiple times for each batch of data, the original parameters that were used to collect the batch are reserved in $\boldsymbol{\theta}_{old}$, which, following similar steps to the ones for (3.2), yields the sample gradient estimate $(G_t^\lambda - \hat{v}_{\mathbf{w}}(S_t))\frac{\nabla \pi_{\boldsymbol{\theta}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}_{old}}(A_t|S_t)}$ in the expected form with sample surrogate objective $-\rho_t(\boldsymbol{\theta})\hat{h}_t$, where the likelihood ratio $\rho_t(\boldsymbol{\theta}) \doteq \frac{\pi_{\boldsymbol{\theta}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}_{old}}(A_t|S_t)}$, and the advantage estimate $\hat{h}_t \doteq (G_t^\lambda - \hat{v}_{\mathbf{w}}(S_t))$. The normalized advantage estimates $\tilde{h}_t$ with zero mean and unit standard deviation are used instead of $\hat{h}_t$ in the algorithm.

After the above changes, PPO modifies its sample surrogate objective to

$$\begin{aligned}
L_{\boldsymbol{\theta},t} &\doteq -\min\left(\rho_t(\boldsymbol{\theta})\tilde{h}_t, \rho_t^{\text{clip}}(\boldsymbol{\theta})\tilde{h}_t\right), \\
\rho_t^{\text{clip}}(\boldsymbol{\theta}) &\doteq \text{clip}(\rho_t(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon), \\
L_{\mathbf{w},t} &\doteq \left(G_t^\lambda - \hat{v}_{\mathbf{w}}(S_t)\right)^2
\end{aligned}$$

with ϵ as a hyper-parameter and uses the semi-gradient sample objective $L_{\mathbf{w},t}$ for the value estimate. The gradient of policy and value estimate sample objectives are derived below:

$$\nabla_{\boldsymbol{\theta}} L_{\boldsymbol{\theta},t} = \begin{cases} -\frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}_{old}}(A_t|S_t)}\tilde{h}_t, & \begin{array}{l}\text{if } \rho_t(\boldsymbol{\theta})\tilde{h}_t \leq \rho_t^{\text{clip}}(\boldsymbol{\theta})\tilde{h}_t \\ \text{or } \left(\rho_t(\boldsymbol{\theta})\tilde{h}_t > \rho_t^{\text{clip}}(\boldsymbol{\theta})\tilde{h}_t \text{ and } 1-\epsilon \leq \rho_t(\boldsymbol{\theta}) \leq 1+\epsilon\right)\end{array} \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

$$\nabla_{\mathbf{w}} L_{\mathbf{w},t} = -2\left(G_t^\lambda - \hat{v}_{\mathbf{w}}(S_t)\right)\nabla_{\mathbf{w}}\hat{v}_{\mathbf{w}}(S_t). \tag{3.4}$$

The original paper explains the intuition behind the policy objective $L_{\boldsymbol{\theta},t}$ in more detail. In short, the min and the clip operators in $L_{\boldsymbol{\theta},t}$ seem to allow the policy to get worse with respect to its objective unlimitedly, but only allow it to improve up to a certain threshold determined by ϵ. Perhaps this asymmetry in the treatment of favorable and unfavorable changes to the policy makes the agent explore for longer and helps avoid local minima in the policy space.

The *Learn* function of the PPO algorithm is shown in Algorithm 3.2. It begins by calculating the λ-return G_t^λ and the advantage estimate $\hat{h}_t$ for all b experience samples in the batch. The alternative recursive definition of $\hat{h}_t$

based on the TD error can be used to calculate these values for all samples efficiently with a $O(b)$ time complexity. The advantage estimates are normalized to have zero mean and unit standard deviation. All samples in the batch are then augmented by their normalized advantage estimates and λ-returns and constitute a dataset D of such samples with size b.

The PPO algorithm then minimizes its objectives $L_{\boldsymbol{\theta},t}$ and $L_{\mathbf{w},t}$ by performing stochastic gradient descent (SGD) updates with mini-batches of size m for multiple epochs N on the dataset D. The PPO hyper-parameters $N = 10$ and $\epsilon = 0.2$ remained fixed across all experiments. We used the Adam optimizer (Kingma & Ba 2014) with the same learning rate of 0.0003 for both objectives.

Our PPO implementation does not account for the issue of time limits that was explained in Section 2.2. The states do not include any information about the amount of time remaining in the episode and are thus partially observable and non-Markovian (Pardo et al. 2018). This may limit the applicability of PPO to our RL problem formulation, which assumed that states have the Markov property. It may impact the performance that the agent achieves in our experiments and is a limitation of our study.

3.3 Reparameterization Policy Gradient Methods

An alternative approach for calculating the gradient of the performance measure $J^{\gamma}(\boldsymbol{\theta})$ was introduced by Lan and Mahmood (2021). It begins by reparameterizing the actions into a deterministic function $f(s, \epsilon, \boldsymbol{\theta})$ that is differentiable with respect to the policy parameters $\boldsymbol{\theta}$. The function receives the state s, a sample ϵ from a fixed distribution $p(\epsilon)$, and the policy parameters $\boldsymbol{\theta}$ as arguments. For instance, actions of the policy $\pi_{\boldsymbol{\theta}}(a|s) = \mathcal{N}(\mu_{\boldsymbol{\theta}}, {\sigma_{\boldsymbol{\theta}}}^2)$ can be reparameterized as

$$f(s, \epsilon, \boldsymbol{\theta}) \doteq \mu_{\boldsymbol{\theta}}(s) + \epsilon\, \sigma_{\boldsymbol{\theta}}(s), \qquad p(\epsilon) = \mathcal{N}(0, 1).$$

In this approach, the actions in the performance measure in (3.1) are reparameterized, and the gradient of $J^\gamma(\boldsymbol{\theta})$ is derived next:

$$\begin{aligned}
J^\gamma(\boldsymbol{\theta}) &\doteq \int_s d_0(s) \int_a \pi_{\boldsymbol{\theta}}(a|s) q_{\pi_{\boldsymbol{\theta}}}(s,a)\ da\ ds \\
&= \int_s d_0(s) \int_\epsilon p(\epsilon) q_{\pi_{\boldsymbol{\theta}}}(s, f(s,\epsilon,\boldsymbol{\theta}))\ d\epsilon\ ds \\
\nabla_{\boldsymbol{\theta}} J^\gamma(\boldsymbol{\theta}) &= \int_s d_0(s) \int_\epsilon p(\epsilon) \Big(\nabla_{\boldsymbol{\theta}} f(s,\epsilon,\boldsymbol{\theta}) \nabla_a q_{\pi_{\boldsymbol{\theta}}}(s,a)|_{a=f(s,\epsilon,\boldsymbol{\theta})} \\
&\qquad\qquad\qquad\qquad + \nabla_{\boldsymbol{\theta}} q_{\pi_{\boldsymbol{\theta}}}(s,a)|_{a=f(s,\epsilon,\boldsymbol{\theta})} \Big)\ d\epsilon\ ds,
\end{aligned}$$

where $a = f(s,\epsilon,\boldsymbol{\theta})$. The $\nabla_{\boldsymbol{\theta}} q_{\pi_{\boldsymbol{\theta}}}(s,a)$ term is again repeatedly expanded and unrolled, giving rise to the reparameterization policy gradient theorem (Lan & Mahmood 2021)

$$\nabla_{\boldsymbol{\theta}} J^\gamma(\boldsymbol{\theta}) = \int_s d^{\gamma,\pi_{\boldsymbol{\theta}}}(s) \int_\epsilon p(\epsilon) \nabla_{\boldsymbol{\theta}} f(s,\epsilon,\boldsymbol{\theta}) \nabla_a q_{\pi_{\boldsymbol{\theta}}}(s,a)|_{a=f(s,\epsilon,\boldsymbol{\theta})}\ d\epsilon\ ds.$$

If an agent follows policy $\pi_{\boldsymbol{\theta}}(a|s)$, this gradient can be written in the expected form as

$$\nabla_{\boldsymbol{\theta}} J^\gamma(\boldsymbol{\theta}) = \mathbb{E}_{d^{\pi_{\boldsymbol{\theta}}},\pi_{\boldsymbol{\theta}}} \Big[\gamma^t \nabla_{\boldsymbol{\theta}} f(S_t,\epsilon_t,\boldsymbol{\theta}) \nabla_a q_{\pi_{\boldsymbol{\theta}}}(S_t,a)|_{a=f(S_t,\epsilon_t,\boldsymbol{\theta})} \Big] \tag{3.5}$$

with $\gamma^t \nabla_{\boldsymbol{\theta}} f(S_t,\epsilon_t,\boldsymbol{\theta}) \nabla_a q_{\pi_{\boldsymbol{\theta}}}(S_t,a)|_{a=f(S_t,\epsilon_t,\boldsymbol{\theta})}$ referred to as the reparameterization policy gradient estimate. Similar to (3.2), the expection is now under the undiscounted on-policy state distribution $d^{\pi_{\boldsymbol{\theta}}}$, and the γ^t term has moved into the expectation. Assuming that the action-value estimate is parameterized by $\mathbf{w}_1$, the sample surrogate objective $L_{\boldsymbol{\theta}} \doteq -\gamma^t \hat{q}_{\mathbf{w}_1}(S_t, f(S_t,\epsilon_t,\boldsymbol{\theta}))$ can be minimized instead.

3.4 The Soft Actor-Critic (SAC) Algorithm

The Soft Actor-Critic algorithm was developed by Haarnoja et al. (2018a). It is an off-policy actor-critic method that updates the policy toward the exponential of the action-value function, formulated as the minimization of KL divergence between the two. Later on, Lan and Mahmood (2021) provided an alternative understanding and derivation of the policy update by introducing the reparameterization policy gradient theorem. The robustness and sample efficiency of SAC make it particularly appealing for real-world tasks, where experience is collected in real-time and at a premium (Haarnoja et al. 2018c).

We can arrive at the sample surrogate objective of SAC from that of the reparameterization policy gradient estimate in (3.5) by adding to it an entropy regularization term, dropping the discounting term γ^t, and using two action-value estimates with parameters $\mathbf{w}_1$ and $\mathbf{w}_2$ as proposed by Fujimoto et al. (2018) to eliminate the positive bias discussed by van Hasselt (2010):

$$L_{\boldsymbol{\theta},t} \doteq \alpha \log \pi_{\boldsymbol{\theta}}(f(S_t, \epsilon_t, \boldsymbol{\theta})|S_t) - \min\Big(\hat{q}_{\mathbf{w}_1}(S_t, f(S_t, \epsilon_t, \boldsymbol{\theta})), \hat{q}_{\mathbf{w}_2}(S_t, f(S_t, \epsilon_t, \boldsymbol{\theta}))\Big).$$

We drop the γ^t term to be consistent with the original SAC paper and acknowledge that, as explained by Nota and Thomas (2020), dropping γ^t makes this a biased estimate. For both parameters of the action-value estimates $\mathbf{w}_1$ and $\mathbf{w}_2$, exponentially moving averages $\overline{\mathbf{w}}_1$ and $\overline{\mathbf{w}}_2$ are maintained as the targets to stabilize training (Mnih et al. 2015). The entropy term is also subtracted from the action-value estimate, which leads to the entropy regularized value function

$$V(S'_{t+1}) \doteq \min\Big(\hat{q}_{\overline{\mathbf{w}}_1}(S'_{t+1}, \tilde{A}_{t+1}), \hat{q}_{\overline{\mathbf{w}}_2}(S'_{t+1}, \tilde{A}_{t+1})\Big) - \alpha \log \pi_{\boldsymbol{\theta}}(\tilde{A}_{t+1}|S'_{t+1})$$

that is used for updating the action-value estimate using $\tilde{A}_{t+1} \sim \pi_{\boldsymbol{\theta}}(\cdot|S'_{t+1})$. The sample objectives for action-value estimates and the temperature parameter $\alpha > 0$ are then defined as:

$$\begin{aligned}
L_{\mathbf{w}_1,t} &\doteq \frac{1}{2}\Big(\hat{q}_{\mathbf{w}_1}(S_t, A_t) - \big(R_{t+1} + (1 - T_{t+1})\gamma V(S'_{t+1})\big)\Big)^2, \\
L_{\mathbf{w}_2,t} &\doteq \frac{1}{2}\Big(\hat{q}_{\mathbf{w}_2}(S_t, A_t) - \big(R_{t+1} + (1 - T_{t+1})\gamma V(S'_{t+1})\big)\Big)^2, \\
L_{\alpha,t} &\doteq -\alpha \log \pi_{\boldsymbol{\theta}}(f(S_t, \epsilon_t, \boldsymbol{\theta})|S_t) - \alpha\overline{\mathcal{H}}
\end{aligned}$$

with $\overline{\mathcal{H}} \doteq -|\mathcal{A}|$ as the desired minimum expected entropy. Gradients of all sample objectives are derived below:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} L_{\boldsymbol{\theta},t} = &\alpha \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|S_t)|_{a=f(S_t,\epsilon_t,\boldsymbol{\theta})} \\
&+ \nabla_{\boldsymbol{\theta}} f(S_t, \epsilon_t, \boldsymbol{\theta})\Big(\alpha \nabla_a \log \pi_{\boldsymbol{\theta}}(a|S_t)|_{a=f(S_t,\epsilon_t,\boldsymbol{\theta})} \\
&- \nabla_a \hat{q}_{\mathbf{w}_{\min}}(S_t, a)|_{a=f(S_t,\epsilon_t,\boldsymbol{\theta})}\Big), \text{ where} \qquad (3.6) \\
\mathbf{w}_{\min} \doteq &\operatorname*{argmin}_{\mathbf{w}\in\{\mathbf{w}_1,\mathbf{w}_2\}} \hat{q}_{\mathbf{w}}(S_t, f(S_t, \epsilon_t, \boldsymbol{\theta})),
\end{aligned}$$

$$\nabla_{\mathbf{w}_i} L_{\mathbf{w}_i,t} = \Big(\hat{q}_{\mathbf{w}_i}(S_t, A_t) - \big(R_{t+1} + (1 - T_{t+1})\gamma V(S'_{t+1})\big)\Big)\nabla_{\mathbf{w}_i}\hat{q}_{\mathbf{w}_i}(S_t, A_t) \quad \text{for } i \in \{1, 2\}, \tag{3.7}$$

$$\nabla_\alpha L_{\alpha,t} = -\log \pi_{\boldsymbol{\theta}}(f(S_t, \epsilon_t, \boldsymbol{\theta})|S_t) - \overline{\mathcal{H}}. \tag{3.8}$$

The agent-environment interaction loop of SAC is similar to that of PPO presented in Algorithm 2.1. However, the learning period $U = 1$ in SAC results in a learning update at every interaction, and the batch size $b =$ 1,000,000 is much larger than a typical batch size in PPO. The *Learn* operation uses and updates $\boldsymbol{\theta}$ for its policy, $\alpha > 0$ as its temperature parameter, and the parameters $\mathbf{w}_1$, $\mathbf{w}_2$, $\overline{\mathbf{w}}_1$, and $\overline{\mathbf{w}}_2$ for its action-value estimates.

The *Learn* function of SAC is listed in Algorithm 3.3. It contains a series of g stochastic gradient descent (SGD) updates. Each iteration starts with sampling a mini-batch M of size m uniformly randomly from the buffer B. The objectives are then calculated and minimized by taking a single gradient step for that mini-batch. The sample objective $L_{\mathbf{w}_1,t} + L_{\mathbf{w}_2,t}$ optimizes the parameters of the action-value estimates, and $L_{\boldsymbol{\theta},t}$ and $L_{\alpha,t}$ respectively optimize the policy parameters and the temperature parameter. The hyper-parameters $m = 256$, $g = 1$, and $\tau = 0.005$ remained constant for all of our experiments. We used the Adam optimizer (Kingma & Ba 2014) for all objectives with learning rates set to 0.0003.

Our SAC implementation handles the issue of time limits discussed in Section 2.2 by bootstrapping from states in which the episode ends due to reaching the time limit. It differentiates between terminal states and non-terminal states that end episodes at the time limit. Only the values of terminal states are considered zero and not bootstrapped from. However, this solution was meant for continuing problems that have to maximize the expected return over an indefinite period (Pardo et al. 2018). The goal of our problem formulation was to maximize the expected undiscounted episodic return. This is a limitation of our study and might affect our results since the time limit issue is not correctly addressed for our specific problem formulation.

We explained how the likelihood-ratio and reparameterization policy gradient theorems can be derived from the discounted performance measure. The objectives of PPO and SAC algorithms were then derived from the likelihood-ratio and reparameterization gradient estimates respectively. Using these objectives, we detailed the *Initialize* and *Learn* functions of both PPO and SAC that can be plugged into and form a complete algorithm with the general agent-environment interaction loop from Section 2.4. Next chapter describes the Reacher Task, on which we examine the robustness of baseline hyper-parameter values to different δts.

Algorithm 3.3: SAC Initialize and Learn functions

```
Function Initialize():
    $\Psi \doteq \{$ learning period $U$, batch size $b$, mini-batch size $m$, gradient
      steps $g$, discount factor $\gamma$, temperature parameter $\alpha$, target
      smoothing coefficient $\tau$, parameterized policy $\pi_{\boldsymbol{\theta}}(a|s)$, policy
      mean $\mu_{\boldsymbol{\theta}}(s)$, policy standard deviation $\sigma_{\boldsymbol{\theta}}(s)$, parameterized
      action-value estimates $\hat{q}_{\mathbf{w}_1}(s,a)$, $\hat{q}_{\mathbf{w}_2}(s,a)$, $\hat{q}_{\overline{\mathbf{w}}_1}(s,a)$, $\hat{q}_{\overline{\mathbf{w}}_2}(s,a)$,
      policy learning rate $\eta^{\boldsymbol{\theta}}$, action-value estimate learning rate $\eta^{\mathbf{w}}$,
      temperature learning rate $\eta^{\alpha}$ $\}$
    Denote network parameters by $\Psi.\boldsymbol{\theta}$, $\Psi.\mathbf{w}_1$, $\Psi.\mathbf{w}_2$, $\Psi.\overline{\mathbf{w}}_1$, $\Psi.\overline{\mathbf{w}}_2$, $\Psi.\alpha$
    Initialize parameters $\Psi.\boldsymbol{\theta}$, $\Psi.\mathbf{w}_1$, and $\Psi.\mathbf{w}_2$
    $\Psi.\overline{\mathbf{w}}_1 \doteq \Psi.\mathbf{w}_1$ ; $\Psi.\overline{\mathbf{w}}_2 \doteq \Psi.\mathbf{w}_2$ ; $\Psi.\alpha \doteq 1$
    return $\Psi$

Function Learn($B$, $\Psi$):
    $(\boldsymbol{\theta}, \mathbf{w}_1, \mathbf{w}_2, \overline{\mathbf{w}}_1, \overline{\mathbf{w}}_2, \alpha) \doteq \Psi.\left(\boldsymbol{\theta}, \mathbf{w}_1, \mathbf{w}_2, \overline{\mathbf{w}}_1, \overline{\mathbf{w}}_2, \alpha\right)$
    for each gradient step from 1 to $g$ do
        Sample a mini-batch $M$ of size $m$ uniformly randomly from $B$
        $\mathbf{w}_1 \doteq \mathbf{w}_1 - \eta^{\mathbf{w}} \frac{1}{m} \sum_{\left(S_k, A_k, R_{k+1}, S'_{k+1}, T_{k+1}\right) \in M} \nabla_{\mathbf{w}_1} L_{\mathbf{w}_1,k}$
        $\mathbf{w}_2 \doteq \mathbf{w}_2 - \eta^{\mathbf{w}} \frac{1}{m} \sum_{\left(S_k, A_k, R_{k+1}, S'_{k+1}, T_{k+1}\right) \in M} \nabla_{\mathbf{w}_2} L_{\mathbf{w}_2,k}$
          using (3.7), where
          $L_{\mathbf{w}_1,k} \doteq \frac{1}{2}\Big(\hat{q}_{\mathbf{w}_1}(S_k, A_k) - \big(R_{k+1} + (1 - T_{k+1})\gamma V(S'_{k+1})\big)\Big)^2,$
          $L_{\mathbf{w}_2,k} \doteq \frac{1}{2}\Big(\hat{q}_{\mathbf{w}_2}(S_k, A_k) - \big(R_{k+1} + (1 - T_{k+1})\gamma V(S'_{k+1})\big)\Big)^2,$
          $V(S'_{k+1}) \doteq \min\Big(\hat{q}_{\overline{\mathbf{w}}_1}(S'_{k+1}, \tilde{A}_{k+1}), \hat{q}_{\overline{\mathbf{w}}_2}(S'_{k+1}, \tilde{A}_{k+1})\Big)$
                    $- \alpha \log \pi_{\boldsymbol{\theta}}(\tilde{A}_{k+1}|S'_{k+1}),$
          and $\tilde{A}_{k+1} \sim \pi_{\boldsymbol{\theta}}(\cdot|S'_{k+1}) \doteq \mathcal{N}\left(\mu_{\boldsymbol{\theta}}(S'_{k+1}), \sigma_{\boldsymbol{\theta}}(S'_{k+1})^2\right)$
        $\boldsymbol{\theta} \doteq \boldsymbol{\theta} - \eta^{\boldsymbol{\theta}} \frac{1}{m} \sum_{\left(S_k, A_k, R_{k+1}, S'_{k+1}, T_{k+1}\right) \in M} \nabla_{\boldsymbol{\theta}} L_{\boldsymbol{\theta},k}$
          using (3.6), where
          $L_{\boldsymbol{\theta},k} \doteq \alpha \log \pi_{\boldsymbol{\theta}}(f(S_k, \epsilon_k, \boldsymbol{\theta})|S_k)$
                 $- \min\Big(\hat{q}_{\mathbf{w}_1}(S_k, f(S_k, \epsilon_k, \boldsymbol{\theta})), \hat{q}_{\mathbf{w}_2}(S_k, f(S_k, \epsilon_k, \boldsymbol{\theta}))\Big),$
          $f(S_k, \epsilon_k, \boldsymbol{\theta}) \doteq \mu_{\boldsymbol{\theta}}(S_k) + \epsilon_k\, \sigma_{\boldsymbol{\theta}}(S_k),$  // reparameterization
          and $\epsilon_k \sim \mathcal{N}(0, 1)$
        $\alpha \doteq \alpha - \eta^{\alpha} \frac{1}{m} \sum_{\left(S_k, A_k, R_{k+1}, S'_{k+1}, T_{k+1}\right) \in M} \nabla_{\alpha} L_{\alpha,k}$
          using (3.8), where
          $L_{\alpha,k} \doteq -\alpha \log \pi_{\boldsymbol{\theta}}(f(S_k, \epsilon_k, \boldsymbol{\theta})|S_k) - \alpha\overline{\mathcal{H}},$
          and $\overline{\mathcal{H}} \doteq -|\mathcal{A}|$
        $\overline{\mathbf{w}}_1 \doteq \tau \mathbf{w}_1 + (1 - \tau)\overline{\mathbf{w}}_1$
        $\overline{\mathbf{w}}_2 \doteq \tau \mathbf{w}_2 + (1 - \tau)\overline{\mathbf{w}}_2$
    end
    $\Psi.\left(\boldsymbol{\theta}, \mathbf{w}_1, \mathbf{w}_2, \overline{\mathbf{w}}_1, \overline{\mathbf{w}}_2, \alpha\right) \doteq \left(\boldsymbol{\theta}, \mathbf{w}_1, \mathbf{w}_2, \overline{\mathbf{w}}_1, \overline{\mathbf{w}}_2, \alpha\right)$
    return $\Psi$
```

Chapter 4

Investigating the Baseline PPO Hyper-Parameters at Different Cycle Times

In this chapter, we study the effect of changing the action-cycle time on the performance of the popular Proximal Policy Optimization (PPO) algorithm using its baseline hyper-parameter values. Algorithm hyper-parameters may need to be tuned every time the action-cycle time δt changes in a given task. However, the cost of hyper-parameter tuning can be prohibitive for real-world robots. Our experiments here aim to answer two questions: Are the baseline hyper-parameters of PPO robust to different cycle times? If not, can changing these hyper-parameters for each δt make the algorithm more robust to different δts? We start by describing the task we experiment on and investigate the robustness of the baseline hyper-parameter values of PPO to different δts.

4.1 The Reacher Task

For our experiments, we used the simulated continuous control robotic task known as *ReacherBulletEnv-v0* from the PyBullet physics engine and task suite (Coumans & Bai 2016). This environment comprises a robotic arm with two rotary joints and two equally-sized links that can move freely in a two-dimensional plane. The base joint of the arm is anchored at the centre of the plane and is connected to the elbow joint via the first link. The elbow joint is also connected to the fingertip of the arm through the second link. To prevent

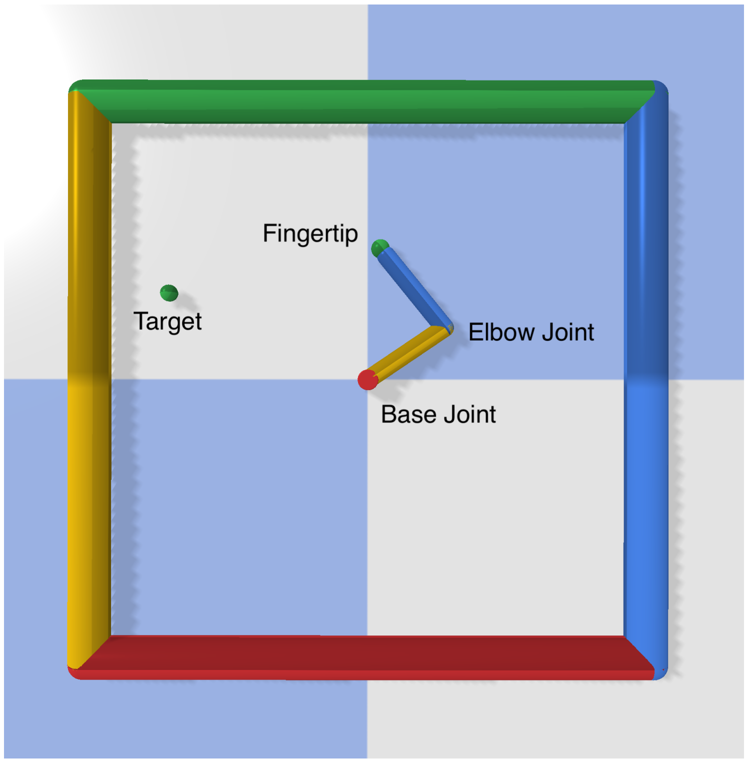

Figure 4.1: The Reacher Task with the robot arm, the fingertip, and the target. The goal is to move the base and elbow joints to get the fingertip as close as possible to the target.

the two links from colliding, the elbow joint can only move in the limited range of $[-3, 3]$ radians.

All episodes last for 2.4 seconds. At the start of each episode, the joints are positioned uniformly randomly in their allowable range and a target is generated in a square area slightly larger than the area reachable by the fingertip. The goal of this task is to move the fingertip to be as close as possible to the target. The observation vector consists of the target position, the vector from the fingertip to the target and the angular positions and velocities of the two joints. The action vector specifies the torque values that are applied to the two joints. In each time step, the task provides three reward components: the *change in distance to the target* to encourage movement of the fingertip toward the target, the *electricity cost* to encourage efficient movement, and the *stuck joint cost* to discourage the elbow joint from getting stuck at either of its limits.

We slightly modify this environment to address two important issues. In the original environment, the angular position of the base joint is calculated relative to its minimum and maximum limits and mapped to the range $[-1, 1]$. The sine and cosine of this relative position are then included in the observation vector. Since the position of the base joint is defined as unlimited, the minimum and maximum values are by default set to 1 and 0 respectively. Taking the sine and cosine of the relative position that has been calculated with respect to those limits results in similar observation vectors for positions that are in fact π radians apart. We changed the observation vector to include the sine and cosine of the absolute angular position instead of the relative ones.

The second issue is related to the third component of the reward. The *stuck joint cost* in the original environment penalizes the agent if the elbow joint is in the vicinity of its limits. Due to the behavior of the physics engine, the elbow joint can go slightly beyond its limits and their vicinities and thus miss the condition for including the mentioned cost in the reward. We modified the condition for the *stuck joint cost* to include the vicinities of and all positions beyond the limits. These modifications improved the performance of PPO with baseline hyper-parameter values significantly. We will open-source the implementation of the task, which can be easily plugged in with existing codebases for experiments.

4.2 Experiments

In this section, we investigate the robustness of baseline hyper-parameter values to different δts for the PPO algorithm. These baseline values can vary for different tasks and environments and can be arrived at in different ways. For instance, they can be tuned to maximize performance on a specific task, adopted from other similar tasks on which they have performed well, or simply set to values recommended by prior works. For our PPO experiments in simulated environments, we use the baseline hyper-parameter values of Table 4.1, which are similar to the recommended hyper-parameters from Schulman et al. (2017) for their Mujoco experiments.

We modify the Reacher Task described in 4.1 by changing its *environment time step* from the default 16.5 ms to a constant 2 ms for all experiments and implementing different action-cycle time δts by making the agent interact with the environment at multiples of 2 ms. To run an experiment with $\delta t =$ 8 ms, for instance, the agent interacts with the environment every fourth environment step, as described in the previous section, by taking the most recent observation as input and outputting an action, which is repeatedly applied to the environment until the next interaction. The reward for each action is a summation over all individual rewards received every environment step of 2 ms.

Each episode still lasts 2.4 simulation seconds for all δts. The environment has three reward components as mentioned earlier: the *change in distance to the target*, the *electricity cost*, and the *stuck joint cost*. The last two are scaled according to the environment time step to keep their weighting relative to the first component comparable to the original environment with 16.5 ms environment time step.

The architecture of the agent comprises a neural network of two hidden layers, each with 64 units and tanh activations, producing the mean μ, and state-independent parameters for the σ of a normal distribution $\mathcal{N}(\mu, \sigma^2)$ from which the actions are sampled. The value estimate is parameterized by another neural network configured similarly to that which produces μ. The Adam optimizer (Kingma & Ba 2014) is used with a learning rate of 0.0003.

To study the robustness of algorithm hyper-parameters to different δts, we ran PPO with the baseline hyper-parameters of Table 4.1 using various δts from 4 to 64 ms and stored the undiscounted episodic returns. The learning curve for each δt is given in Figure 4.2. We further performed a grid search with five different values of the batch size from 500 to 8000 and mini-batch size from 12 to 200 to see if any improvements could at all be made to the learning performance of different δts. The overall average return for each set of hyper-parameters was calculated by averaging the return over the entire learning period and 30 independent runs. The best-performing tuned hyper-parameters for each δt were then used to perform another set of independent

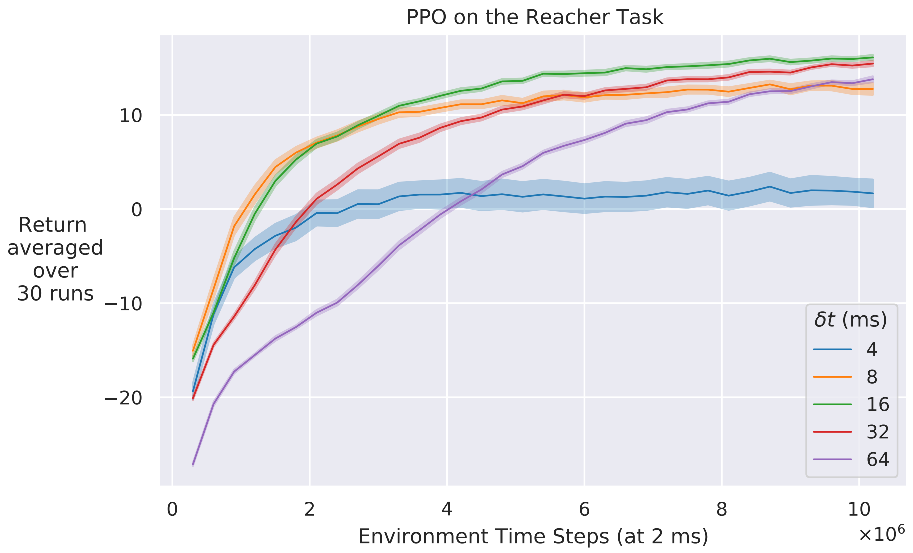

Figure 4.2: Learning curves for different δts with the baseline hyper-parameters. Smaller δts have worse asymptotic performance while larger δts learn more slowly.

runs to avoid maximization bias, and the results were plotted in Figure 4.3. Both plots show results that use undiscounted returns and are averaged over 30 independent runs each lasting 10 million environment steps. The shaded area represents the standard error.

baseline hyper-parameters	value
b	2000
m	50
γ	0.99
λ	0.95

Table 4.1: Baseline Hyper-Parameters of PPO on the Reacher and the Double Pendulum Tasks.

4.3 Results and Discussion

Our experimental results indicate that the baseline hyper-parameters can lead to suboptimal performance when changing the cycle time, and that performance can be recovered, at least partially, by using a different set of hyper-parameters. Figure 4.2 shows that the asymptotic performance declines when

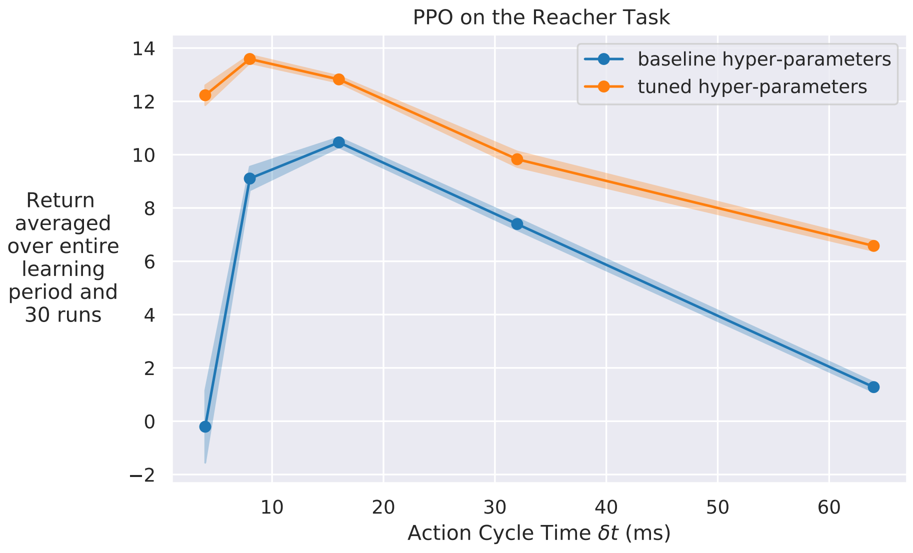

Figure 4.3: Overall average return vs. δt for different hyper-parameter configurations. Tuned hyper-parameters can improve the learning performance for the majority of δt values.

using small δts and that large δts hurt the learning speed. The former may be because batches are collected more quickly, lacking enough information for useful updates. The latter might be a result of fewer and infrequent updates. We show in Figure 4.3 that tuning hyper-parameters for each δt can lead to increased learning performance. The tuned batch size and mini-batch size values are different from the baseline values and shown in Table 4.2. The performance for the tuned hyper-parameters exhibits a general downward trend as δt is increased, which could hint at the superior performance of smaller δts on this task.

tuned	δt (ms)				
hyper-parameters	4	8	16	32	64
b	8000	8000	4000	2000	1000
m	50	25	12	12	12

Table 4.2: Tuned Hyper-Parameters of PPO on the Reacher Task.

We have answered the two questions formed in the beginning of the chapter. We showed that the baseline hyper-parameter values of PPO are not robust to different δts and changing the hyper-parameter values for each δt can perform better than the baseline values. Hence, algorithm hyper-parameters may need to be tuned every time the δt of a task is changed. This demonstrates the importance of having guidelines for adjusting different hyper-parameters based on δt to enable hyper-parameters tuned to a specific δt to be transferred to a different δt on the same task.

Chapter 5

Setting Hyper-Parameters of PPO as a Function of the Action-Cycle Time

This chapter introduces our proposed new hyper-parameters for the Proximal Policy Optimization (PPO) algorithm that adapt to different action-cycle time δts. The baseline hyper-parameter values of PPO are not robust to different δts as we demonstrated in the previous chapter. Although the baseline hyper-parameter values impaired the performance of PPO when δt changed, tuning them partially regained the lost performance. Accordingly, every time a new δt is tried on a real-world robotic task, algorithm hyper-parameters should be laboriously tuned. We aim to avoid this inconvenience by providing guidelines for adjusting the hyper-parameters based on δt as it changes and hypothesize that they can perform better than or as well as the baseline hyper-parameter values. In the following, we describe these guidelines and study their effectiveness compared with the baselines.

5.1 Our Proposed δt-Aware Hyper-Parameters of PPO

Changing the cycle time may affect the relationship between time, and the hyper-parameters batch size b and mini-batch size m, which could suggest the need for scaling them based on δt. As δt decreases, each fixed-size batch or mini-batch of samples corresponds to less amount of real-time experience, possibly reducing the extent of useful information that is available in the batch or the mini-batch.

To address this, we make the real-time experience content consistent among different δts by scaling the baseline b and m inversely proportionally to δt, shown in (5.1) as $b_{\delta t}$ and $m_{\delta t}$. The scaled $b_{\delta t}$ and $m_{\delta t}$ represent the batch size and mini-batch size used when δt is the action-cycle time (in ms). For instance, when δt is reduced from the baseline $\delta t_0 = 16$ ms to 8 ms, a batch size b_8 double the size of the baseline b_{16} is used: $b_8 = 2b_{16}$. This keeps the *batch time* $\delta t \cdot b_{\delta t}$, the time it takes to collect a batch, and the amount of available information consistent across different δts.

The cycle time may influence the choice of the discount factor γ and the trace-decay parameter λ as well since it changes the rate at which rewards and n-step returns are discounted through time (Doya 2000, Tallec et al. 2019). When using a small δt, rewards are discounted more heavily for the same γ and λ, as more experience samples are collected in a fixed time interval compared to larger δts.

Based on the above intuition discussed in previous works (Baird 1994, Doya 2000), we exponentiated γ and λ to the $\delta t/\delta t_0$ power with baseline action-cycle time δt_0. However, our experiments revealed this strategy to be detrimental to the performance of smaller δts. We conjecture that with this strategy, as δt gets smaller, an increasing number of samples are included in the calculation of the likelihood-ratio policy gradient estimate, possibly leading to its increased variance (Munos 2006) and hindered performance. Hence, we only exponentiate γ and λ to the $\delta t/\delta t_0$ power for δts larger than δt_0. This can be achieved by setting $\gamma_{\delta t}$ and $\lambda_{\delta t}$ to be the minimum of the baseline and the scaled one as shown in (5.1). Based on the mentioned modifications, we present the new *δt-aware hyper-parameters* $b_{\delta t}$, $m_{\delta t}$, $\gamma_{\delta t}$ and $\lambda_{\delta t}$, which adapt to different δts according to:

$$\begin{gathered} b_{\delta t} \doteq \frac{\delta t_0}{\delta t} b_{\delta t_0}, \ \ m_{\delta t} \doteq \frac{\delta t_0}{\delta t} m_{\delta t_0}, \\ \gamma_{\delta t} \doteq \min\left(\gamma_{\delta t_0}, \gamma_{\delta t_0}^{\delta t/\delta t_0}\right), \ \ \lambda_{\delta t} \doteq \min\left(\lambda_{\delta t_0}, \lambda_{\delta t_0}^{\delta t/\delta t_0}\right), \end{gathered} \tag{5.1}$$

where δt_0 is the baseline δt.

5.2 Experiments

We evaluated the δt-aware hyper-parameters of Table 5.1 ($\delta t_0 = 16$ ms) by running PPO on the same Reacher Task from the previous section using different batch times (batch size $\times$ δt) ranging from 1 to 128 seconds. The cycle times used were similar to the previous chapter, and each run continued for 10 million environment steps. Figure 5.1 depicts the average learning curves from 30 independent runs for different batch times. To better understand how changing the batch time affected the performance, we calculated the overall average return by averaging the return over the entire learning period and the 30 runs and plotted it against the batch times in Figure 5.2.

hyper-parameters		value
baseline	δt-aware	
b	b_{16}	2000
m	m_{16}	50
γ	γ_{16}	0.99
λ	λ_{16}	0.95

Table 5.1: Baseline and δt-Aware Hyper-Parameters of PPO on the Reacher and the Double Pendulum Tasks.

The same experiment was repeated with $b_{\delta t}$ and $m_{\delta t}$ from (5.1), but using $\gamma_{\delta t} = \gamma_{\delta t_0}^{\delta t/\delta t_0}$ and $\lambda_{\delta t} = \lambda_{\delta t_0}^{\delta t/\delta t_0}$ unrestrictedly instead, and the learning curves were compared with the δt-aware ones in Figure 5.3. Once again, the average of each learning curve, or the overall average return, was drawn for different batch times in Figure 5.4. To compare the above results with the baseline γ and λ, we repeated the same experiment using (5.1) but kept $\gamma_{\delta t} = \gamma_{16} = 0.99$ and $\lambda_{\delta t} = \lambda_{16} = 0.95$ constant across all runs. The corresponding learning curves and their overall average return summary were compared with the δt-aware ones in Figures 5.5 and 5.6 respectively. All results are averaged over 30 independent runs and use undiscounted returns. The shaded region in all figures shows the standard error.

5.3 Results and Discussion

The learning curves of Figure 5.1 show great disparity between different δts for the two smallest batch times. This disparity subsides as the batch time increases and comes back for the larger batch times, albeit less severely. The asymptotic performance consistently improves with increasing batch times, achieving values close to the maximum from the 32 second batch time onward. This improvement may be attributed to the increasing batch time since it can reduce the variance in the updates of the algorithm by having more samples in a batch. The largest $\delta t = 64$ ms does not attain the peak performance obtained by other δts possibly hinting at the advantage of smaller δt in achieving superior performance. Larger δts seem to suffer from lower learning speeds with the largest batch time, which might be due to smaller δts's having more experience samples available in each batch.

Figure 5.2 aggregates the learning curves of Figure 5.1. It illustrates that the δt-aware hyper-parameters make the performance of small δts better than or equal to that of large δts at larger batch times and thus showcases the benefit of small δts over larger ones in this task with our δt-aware hyper-parameters. Our δt-aware suggestion of keeping the batch time constant when changing δts is supported by the fact that the curves for different δts are aligned together with respect to the batch time. Choosing a constant batch time of 64 seconds can lead to near-best performance for all δts. Performance can rapidly deteriorate if the batch time is reduced to lower than 24 seconds. Keeping the batch size constant at 2000 equates to insufficient batch times for the smaller δts and thus particularly hurts their performance. The circles with dashed borders show the performance of baseline hyper-parameters. The δt-aware hyper-parameters of Table 5.1 are an improvement over the baseline values as shown by the solid-bordered circles at the batch time of 32 seconds.

In the second set of experiments, the discount factor and the trace-decay parameter were changed from above to $\gamma_{\delta t} = \gamma_{\delta t_0}^{\delta t/\delta t_0}$ and $\lambda_{\delta t} = \lambda_{\delta t_0}^{\delta t/\delta t_0}$ (scaled). In Figure 5.3, only the $\delta t = 4$ and $\delta t = 8$ ms learning curves are changed compared to Figure 5.1 and therefore plotted. This is expected since the

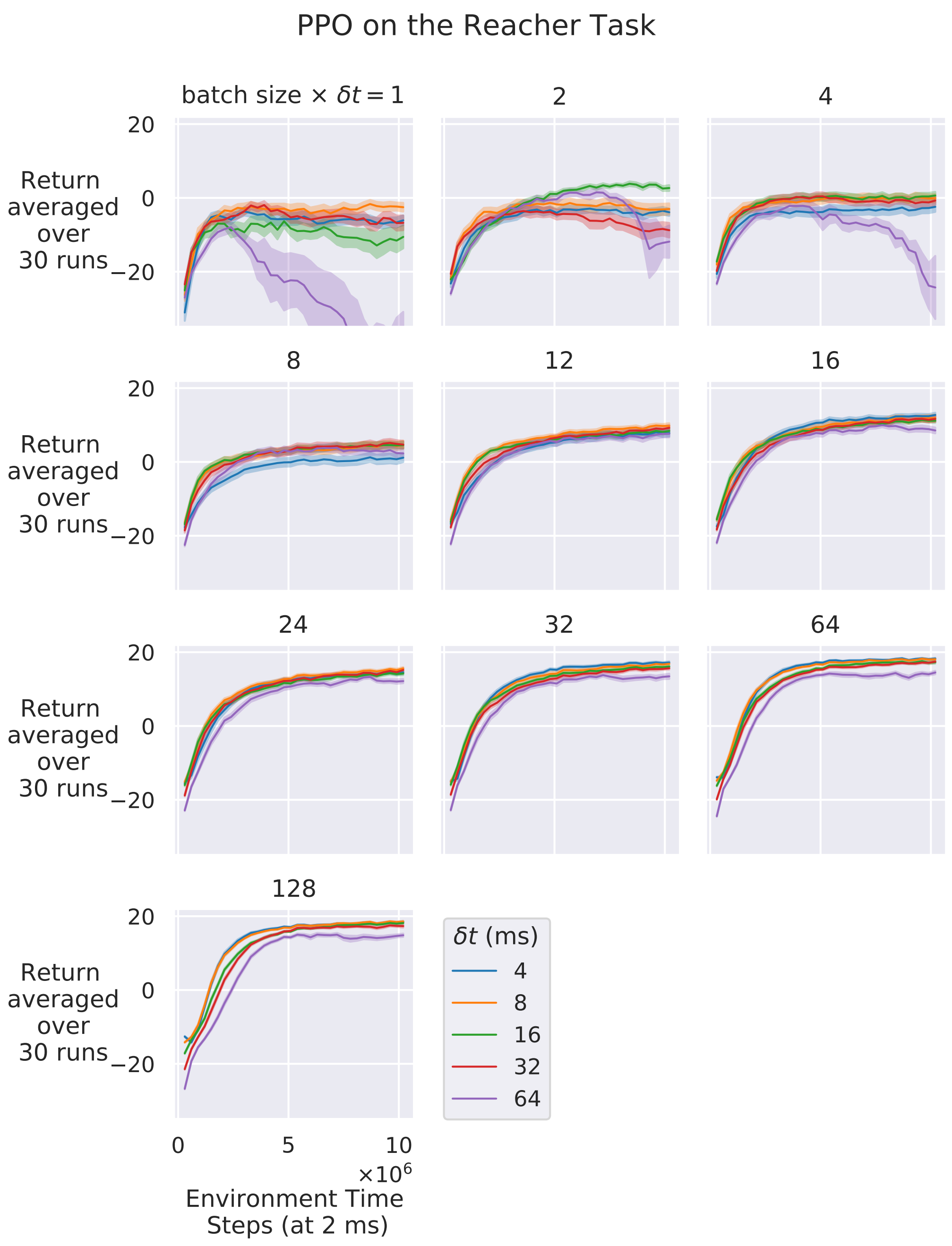

Figure 5.1: Learning curves using the δt-aware hyper-parameters. Performance improves with increasing batch times for all δts.

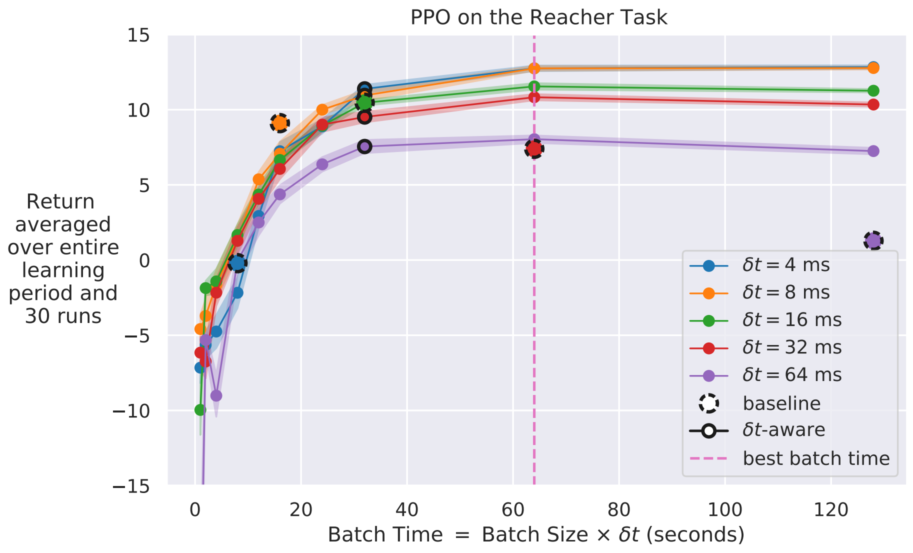

Figure 5.2: Overall average return using the δt-aware hyper-parameters. Small δts are better than or equal to large δts at larger batch times. The δt-aware hyper-parameters at 32-second batch time perform better than the baselines.

hyper-parameters have the same value for the other three δts because of the minimum function in (5.1). The scaled $\delta t = 4$ and $\delta t = 8$ ms learning curves still show similar traits to the δt-aware ones. Their asymptotic performance is improved with increasing batch time. For larger batch times, however, the scaled $\delta t = 4$ and $\delta t = 8$ ms consistently perform worse than the δt-aware.

Figure 5.4 summarizes Figure 5.3 wherein just the $\delta t = 4$ and $\delta t = 8$ ms curves are drawn to compare the δt-aware and scaled γ and λ. It shows that exponentiating γ and λ to the $\delta t/\delta t_0$ power for all δts leads to reduced performance for smaller δts at larger batch times compared to our δt-aware hyper-parameters. We suspect that using a smaller δt increases the variance of the updates (Munos 2006), which is possibly alleviated in δt-aware γ and λ by discounting faster through time.

For the third set of experiments, we experimented with using (5.1), but keeping γ_{16} and λ_{16} constant across all δts (constant). As expected, only the learning curves of $\delta t = 32$ and $\delta t = 64$ ms are drawn in Figure 5.5 since the other three δts have the same hyper-parameter values and are similar to Figure 5.1. The asymptotic performance of constant γ and λ still improves

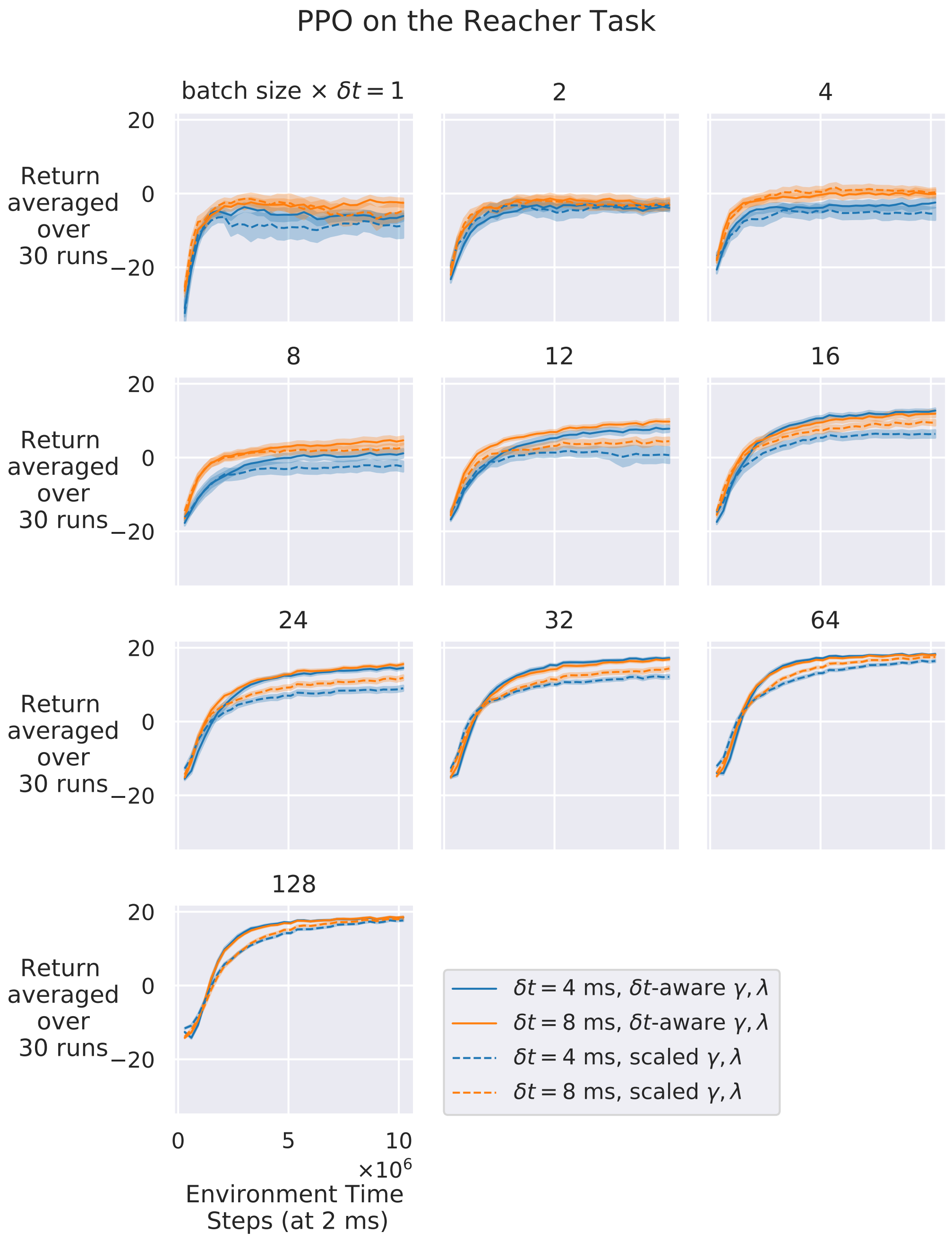

Figure 5.3: Learning curves comparing the δt-aware hyper-parameters with ones where the discount factor γ and trace-decay parameter λ are always exponentiated to the $\delta t/\delta t_0$ power (scaled). Only the $\delta t = 4$ ms and $\delta t = 8$ ms curves are plotted since the rest are similar to Figure 5.1.

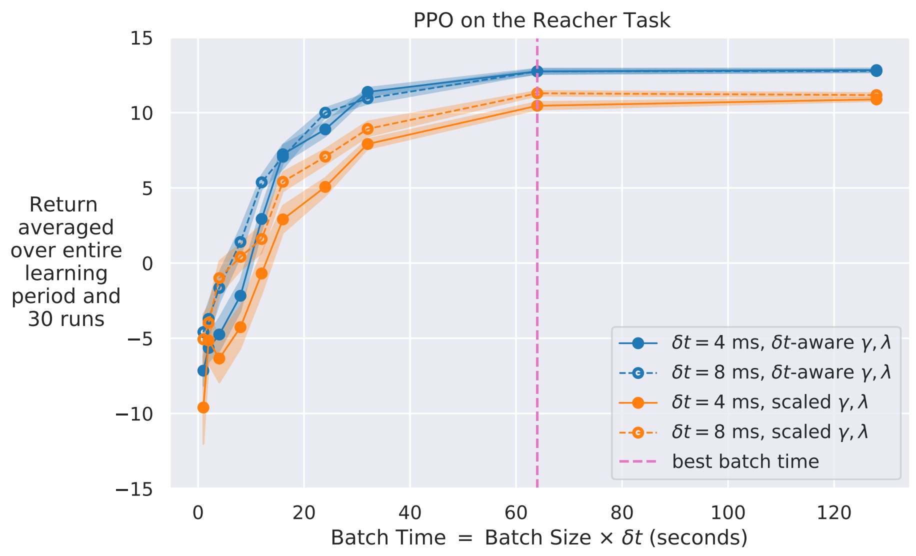

Figure 5.4: Overall average return of δt-aware hyper-parameters compared with ones where γ and λ are always exponentiated to the $\delta t/\delta t_0$ power (scaled).

with increasing batch time, and the learning speed is worse than the δt-aware at the largest batch time. Although, the performance of $\delta t = 32$ and $\delta t = 64$ ms with constant γ and λ is diminished for the same batch time when compared with the δt-aware ones.

The learning curves of Figure 5.5 are summarized in Figure 5.6, and $\delta t = 32$ and $\delta t = 64$ ms perform worse compared with our δt-aware hyper-parameters. Keeping the discount factor and the trace-decay parameter constant for larger δts results in these hyper-parameters attenuating more slowly in real-time. This might increase the variance of the updates since more real-time experience is included in their calculation, and might have been avoided in the δt-aware ones by keeping the mentioned real-time experience constant for the updates.

We presented the δt-aware hyper-parameters of PPO that can adapt their values to different δts and empirically demonstrated that they can perform better than or equal to the baseline hyper-parameters. Overall, our results indicate that the performance of different δts show similar sensitivity to the batch time or amount of real-time experience in each batch, warranting our recommendation for keeping batch time constant as δt changes. Smaller δts perform better if γ and λ decay faster than the baseline through time. Larger

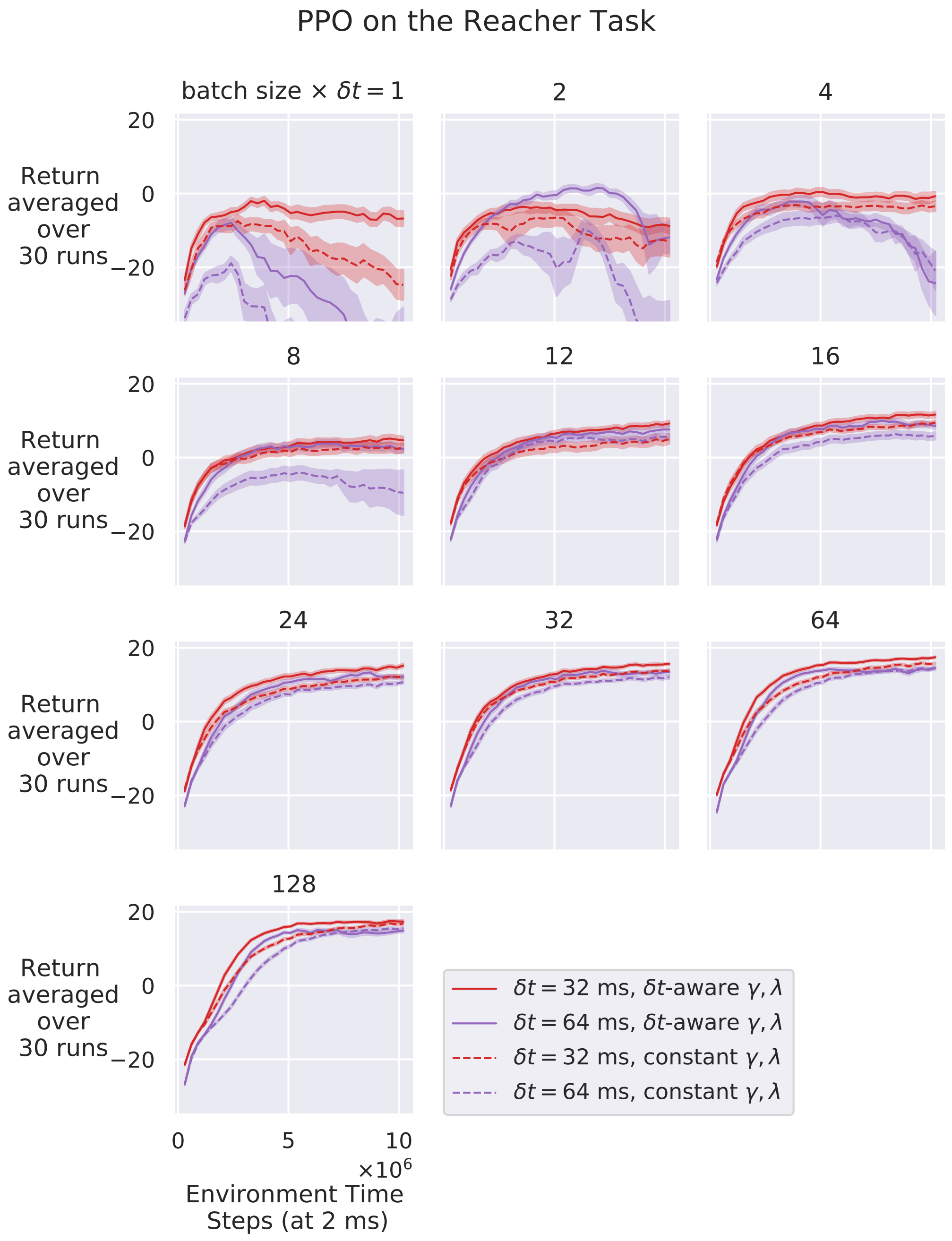

Figure 5.5: Learning curves comparing the δt-aware hyper-parameters with ones where the discount factor $\gamma_{\delta t} = \gamma_{16} = 0.99$ and trace-decay parameter $\lambda_{\delta t} = \lambda_{16} = 0.95$ are constant in all runs (constant). Only the $\delta t = 32$ ms and $\delta t = 64$ ms curves are plotted since the rest are similar to Figure 5.1.

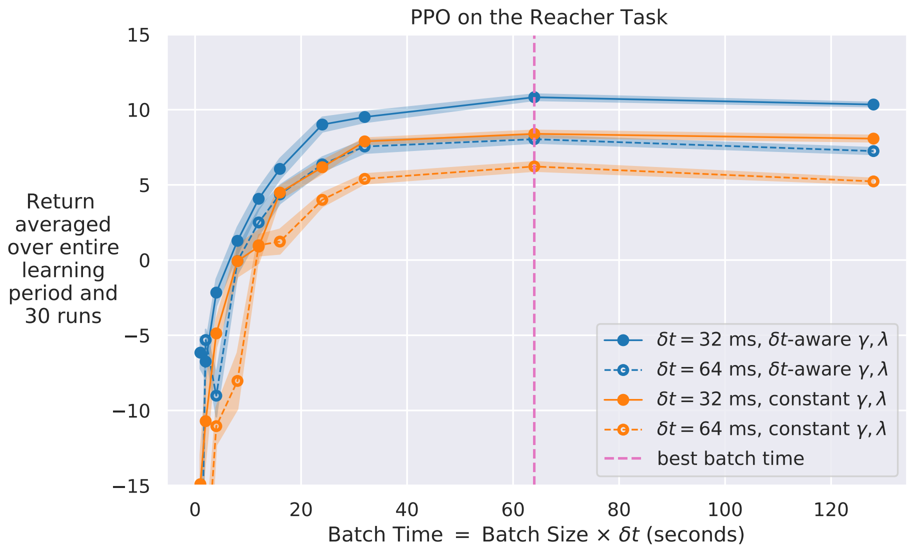

Figure 5.6: Overall average return of δt-aware hyper-parameters compared with ones with constant $\gamma_{\delta t} = \gamma_{16} = 0.99$ and $\lambda_{\delta t} = \lambda_{16} = 0.95$ in all runs.

δts perform worse if γ and λ decay slower than the baseline through time. The δt-aware γ and λ take these observations into account. We validate the δt-aware hyper-parameters on two previously unseen tasks in the next chapter.

Chapter 6

Validating the δt-Aware Hyper-Parameters of PPO

In this chapter, we validate our proposed δt-aware hyper-parameters for the Proximal Policy Optimization (PPO) algorithm on two previously unseen tasks. The δt-aware hyper-parameters are advantageous compared to the baselines on the original Reacher Task as demonstrated earlier. However, the robustness and applicability of these δt-aware hyper-parameters to different environments and particularly real-world tasks remains a question that we explore here. We describe the two simulated and real-world robotic tasks, on which we then compare the performance of the δt-aware hyper-parameters with the baseline values.

6.1 The Double Pendulum Task

The simulated PyBullet task *InvertedDoublePendulumBulletEnv-v0* is a continuous control robotic task akin to two-dimensional pole balancing. The robot consists of two links, two non-actuated free-moving rotary joints, and a slider joint that can move within a limited range along a horizontal axis. The first rotary joint connects the slider joint to the first link, which is in turn connected to the second link through the second rotary joint. The fingertip resides on the other side of the second link.

Episodes of this task last for a maximum of 16 seconds. At the beginning of each episode, the two links (the pole) are set to a random almost upright position just as gravity starts to pull them down. The episode can be terminated early if the fingertip falls below a certain height. The goal then is to move the slider joint to keep the two links as upright as possible. The observation vector includes the positions and velocities of all three joints in addition to the horizontal position of the fingertip. The one-dimensional action sets the torque of the slider joint. At each time step, the agent is rewarded with a constant value and penalized proportionally to the distance of the fingertip from the upright and horizontally centred position.

6.2 Validation on Double Pendulum

We validated the δt-aware hyper-parameters of PPO on the Double Pendulum Task from the previous section. We reduced the environment time step of this PyBullet environment from 16.5 ms to a constant 4 ms and simulated other δts as integer multiples of 4 ms. Episodes of this environment still lasted a maximum of 16 simulation seconds for all δts. All results were averaged over 30 independent runs and use undiscounted returns. Shaded regions indicate standard errors.

We ran two sets of experiments with PPO. For the first set, we kept the baseline hyper-parameters of Table 5.1 ($b = 2000$) constant across δts, and for the second set, we used the δt-aware hyper-parameters of Table 5.1 with $\delta t_0 = 16$ ms ($b_{16} = 2000$). All runs continued for 10 million environment steps, and the learning curves for these two sets using the baseline and δt-aware hyper-parameters were plotted in Figures 6.1 and 6.2 respectively. We further calculated the average return over the full learning period for each learning curve with results shown in Figure 6.3 for both sets of experiments.

The learning curves of Figure 6.1 demonstrate that the baseline hyper-parameters of PPO lead to diminished asymptotic performance as δt decreases. For smaller δts, each fixed-size batch corresponds to less amount of real-time experience, which might not be enough to make useful updates. In contrast, larger δts seem to suffer from slower learning speeds with the baseline hyper-parameters. As δt increases, it takes more time to collect each batch. This

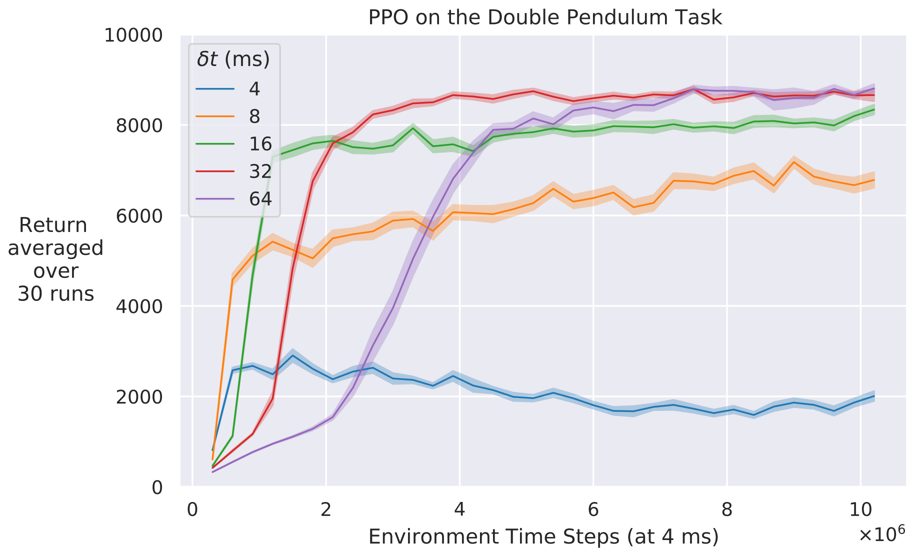

Figure 6.1: Learning curves using the baseline hyper-parameters on the Double Pendulum Task. Different δts vary greatly in asymptotic performance and learning speed.

results in fewer updates being made per real-time and possibly slower learning. The δt-aware hyper-parameters improve the asymptotic performance of smaller δts and slightly reduce that of larger δts compared with the baseline values as shown in Figure 6.2. The learning speeds of all δts are also more comparable with the δt-aware hyper-parameters as opposed to the baseline ones.

Figure 6.3 summarizes the previous two figures by showing the return averaged over the entire learning period and the 30 runs against δt for both baseline and δt-aware hyper-parameters. The δt-aware hyper-parameters provide a significant improvement over the baseline values and make the performance of the algorithm more robust to different δts. These hyper-parameters attempt to make the asymptotic performance and the learning speed of different δts similar to each other as evident by the corresponding learning curves. The baseline and the δt-aware hyper-parameters have equivalent values at $\delta t = 16$ ms and thus exhibit the same performance for this particular δt.

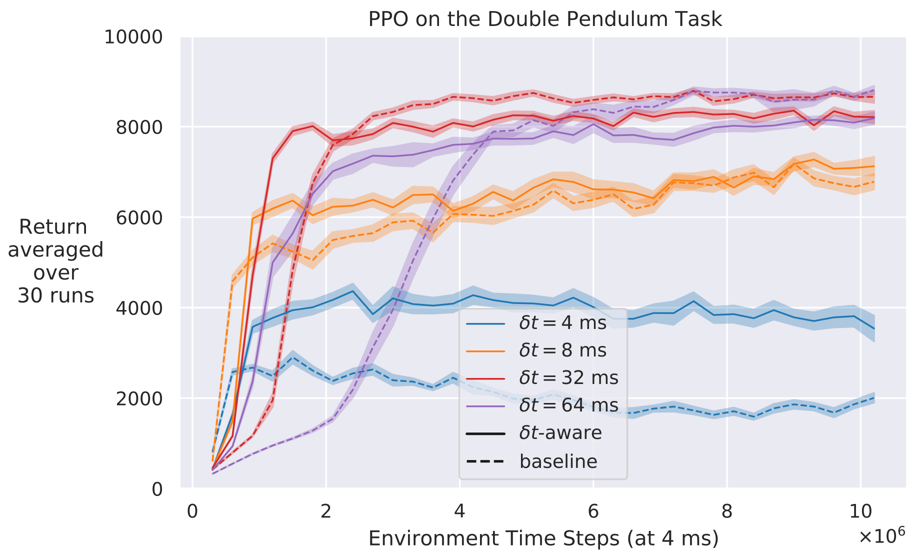

Figure 6.2: Learning curves comparing the baseline and δt-aware hyper-parameters on the Double Pendulum Task. With the δt-aware hyper-parameters, different δts are more similar in asymptotic performance and learning speed.

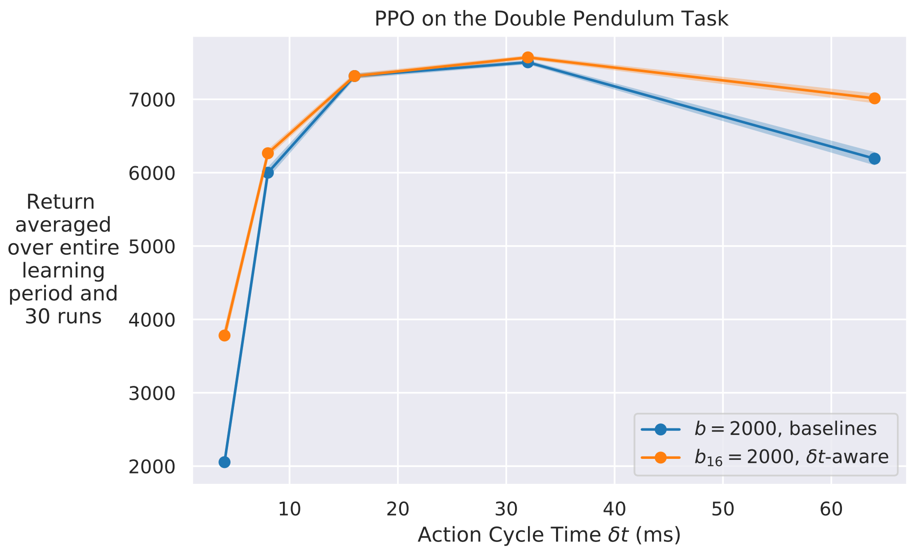

Figure 6.3: Overall average return for different δts on a simulated validation task. The δt-aware hyper-parameters make the performance more robust to different choices of δt compared with the baselines.

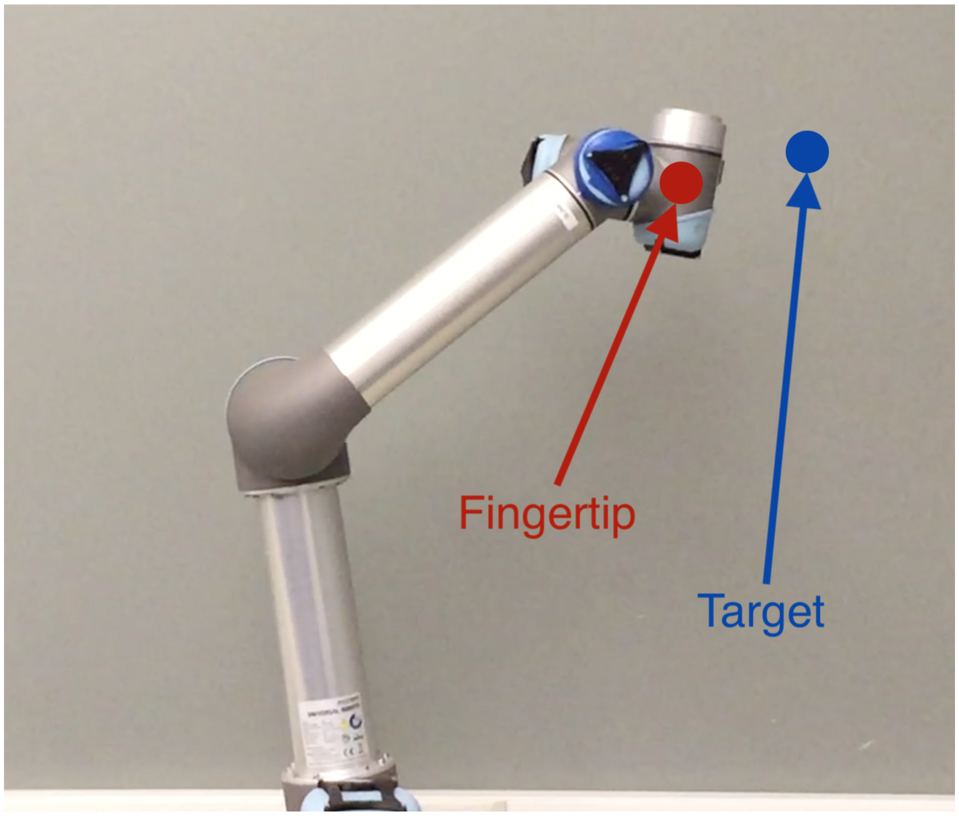

Figure 6.4: The Real-Robot Reacher Task with the robot arm, the fingertip, and the target. The goal is to move the base and elbow joints to get the fingertip as close as possible to the target.

6.3 The Real-Robot Reacher Task

Our real-world robotic validation task is *UR-Reacher-2* developed by Mahmood et al. (2018b). It is embodied via a robot arm that has two rotary joints and two links. The base joint is fixed to a horizontal table and connected to the elbow joint through the first link. The second link connects the elbow joint to the fingertip. We have slightly modified the task so that the arm can move freely in a two-dimensional plane above the table as long it does not collide with itself or the table. In the original environment, the fingertip was confined to a small rectangular area, which caused unnecessary scripted position corrections. This environment is depicted in Figure 6.4 with the robot arm and its fingertip.

Each episode lasts for 4 seconds, at the start of which the arm is reset to a fixed default position, and a target is generated uniformly randomly in a rectangular area above the table. The goal is to move the arm so that the fingertip is as close to the target as possible. The observation vector consists of the angular position and velocity of the two joints, the vector from the fingertip to the target, and the previous action. The action vector sets the velocities of the two joints. The agent is negatively rewarded proportionally to the distance of the fingertip to the target and positively rewarded according to the position of the fingertip in a Gaussian function centred on the target.

6.4 Validation on Real-Robot Reacher

We validated the δt-aware hyper-parameters of PPO on the Real-Robot Reacher Task described in the previous section. We set the environment time step to 10 ms and ran three sets of experiments using PPO, one with $\delta t_0 = 40$ ms as the benchmark, and two with $\delta t = 10$ ms to compare the baseline hyper-parameters of Table 6.1 ($b = 400$), kept constant across δts, to the δt-aware hyper-parameters that adapt to $\delta t = 10$ ms using $b_{10} = \frac{\delta t_0}{10} \cdot b_{\delta t_0} = \frac{40}{10} \cdot 400 = 1600$. All episodes were still 4 seconds long for all δts. Each run lasted for 600,000 environment steps or 100 real-time minutes, and the learning curves, averaged over five runs, were plotted in Figure 6.5. We performed similar experiments using baseline and δt-aware hyper-parameter values of Table 5.1 ($b = 2000$) with the results shown in Figure 6.6. All results use undiscounted returns. Shaded regions represent the standard error.

Figure 6.5 shows that the δt-aware hyper-parameters of PPO on the real-world robotic task recover the asymptotic performance that is lost by using the baseline hyper-parameters with a small δt. Using the baseline hyper-parameters at $\delta t = 10$ ms significantly impairs both the learning speed and the asymptotic performance compared to $\delta t_0 = 40$ ms, possibly because each batch corresponds to less amount of real-time experience. The δt-aware hyper-parameters at $\delta t = 10$ ms can achieve an asymptotic performance similar to or slightly better than the baselines at $\delta t_0 = 40$ ms. The learning speed of the δt-aware hyper-parameters at $\delta t = 10$ ms is slightly worse than the baselines at $\delta t_0 = 40$ ms, which could hint at the increasing difficulty of solving tasks with smaller δts.

When using the hyper-parameter values of Table 5.1 ($b = 2000$), the performances of $\delta t = 10$ ms and $\delta t_0 = 40$ ms are comparable when using the baseline hyper-parameters as illustrated in Figure 6.6. This is likely due to the fact that a batch size of 2000 is already large enough for the algorithm to make useful learning updates at $\delta t = 10$ ms. The δt-aware hyper-parameters can still attain an asymptotic performance similar to the baseline values, though with slightly lower learning speed. In any case, the benefit of the δt-aware hyper-parameters is more pronounced for the hyper-parameter values of Table 6.1, which are more likely to be chosen as the baseline values since they learn faster in Figure 6.5.

hyper-parameters		value
baseline	δt-aware	
b	b_{40}	400
m	m_{40}	10
γ	γ_{40}	0.99
λ	λ_{40}	0.95

Table 6.1: Hyper-Parameters of PPO on the Real-Robot Reacher Task.

The experiments of this chapter validated the δt-aware hyper-parameters of PPO on a simulated and a real-world robotic task. On both tasks, the δt-aware hyper-parameters performed better than or as well as the baseline values. The results are particularly encouraging for the real-world task since hyper-parameter values that have been tuned for a specific δt on a task may be transferred to different δts on the same task, possibly avoiding the laborious and costly tuning of hyper-parameters each time δt is changed.

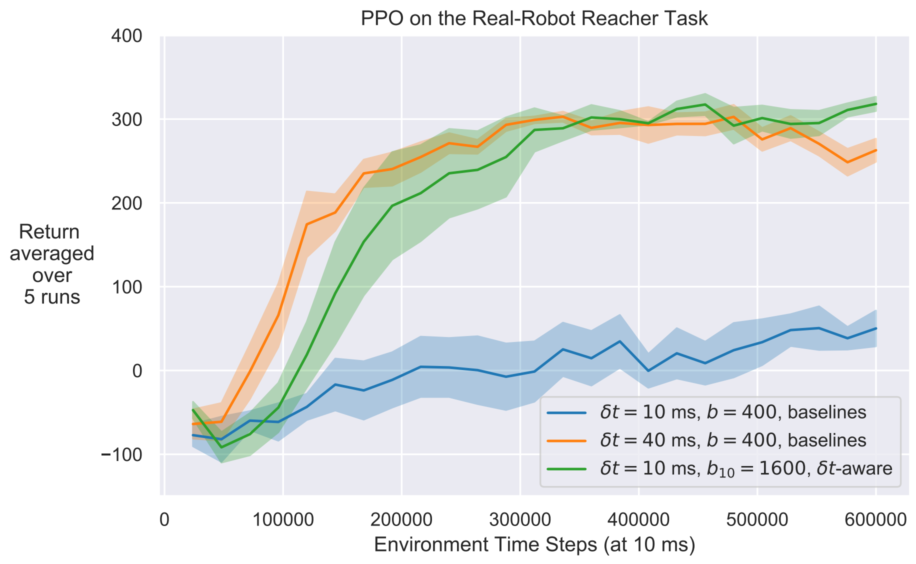

Figure 6.5: Learning curves of PPO on the Real-Robot Reacher Task comparing the δt-aware hyper-parameters of Table 6.1 with the baseline ones. Asymptotic performance is recovered by the δt-aware hyper-parameters for the smaller δt.

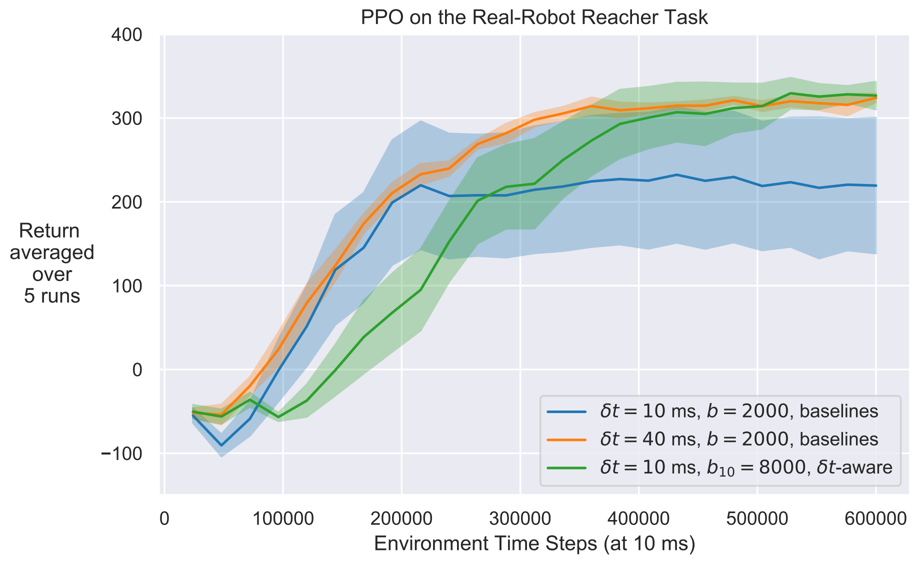

Figure 6.6: Learning curves of PPO on the Real-Robot Reacher Task comparing the δt-aware and baseline hyper-parameters of Table 5.1. Performance of $\delta t = 10$ ms and $\delta t_0 = 40$ ms are comparable using the baseline values.

Chapter 7

Investigating the Discount Factor of SAC at Different Cycle Times

This chapter investigates the robustness of the baseline hyper-parameter values of the Soft Actor-Critic (SAC) algorithm to different action-cycle time δts and examines the sensitivity of performance to different values of the discount factor γ. If the baseline values are not robust to different δts, the algorithm hyper-parameters need to undergo costly and time-consuming tuning every time δt changes in a task. Our goal here is to find out whether the baseline hyper-parameter values of SAC hinder its performance for different δts, and if yes, whether adjusting γ based on δt can make the performance of SAC better than or equal to that of the baseline γ. We examine the robustness of the baseline hyper-parameter values of SAC to different δts and perform an exhaustive sweep over the discount factor γ to better understand its relationship with the action-cycle time.

7.1 Experiments

In this section, we examine how changing δt influences the performance of SAC and how scaling the discount factor γ based on δt can affect the performance for different δts. We started by investigating the robustness of the baseline hyper-parameters of SAC to different cycle times. Both the policy and the action-value estimates were parameterized using neural networks with two hidden layers of size 256 with ReLU activations. The hidden layers of the policy parameters were shared between mean μ and standard deviation σ of the normal distribution $\mathcal{N}(\mu, \sigma^2)$ that was used to draw the actions. We used the Adam optimizer with a learning rate of 0.0003.

On the Reacher Task described in Section 4.1, we learned policies at different δts using the SAC algorithm with automatic entropy adjustment and the hyper-parameter values given by Haarnoja et al. (2018b) as our baseline. All runs lasted for 500,000 environment steps of 2 ms each, and all other experimental details were kept as in the previous sections. Figure 7.1 shows the resulting learning curves for each δt, averaged over 30 independent runs.

We explained previously how changing δt can cause the rewards to be discounted faster or slower through time if γ is kept constant. As such, we continued by studying the effect of scaling and tuning the discount factor γ on performance. We scaled the discount factor for each δt according to $\gamma_{\delta t} = \gamma_{16}^{\delta t/16}$ with baseline $\gamma_{16} = 0.99$ to ensure consistent discounting of the rewards through time regardless of δt (Doya 2000, Tallec et al. 2019). The resulting returns were then averaged over the entire learning period and 30 runs and plotted in Figure 7.2. Similar curves were drawn for two additional sets of experiments. One where a baseline $\gamma = 0.99$ stayed constant across all δts, and another where the best $\gamma_{\delta t}$ was found by searching over 12 different values from 0.2763 to 1.0 for each δt individually and rerun to avoid maximization bias. All results use undiscounted returns. The shaded region in all plots shows the standard error.

7.2 Results and Discussion

As evident in Figure 7.1, the smaller cycle times $\delta t = 4$ and $\delta t = 8$ ms undergo sharp drops in performance just as the agent starts using the learned policy, although they are able to recover and match other δts swiftly. The larger $\delta t = 64$ ms sustains a slower rate of learning throughout and never quite exceeds the learned performance of any other δt. Larger δts obtain lower average returns at the start of learning when the agent acts according to a random policy. This could be due to an increased electricity cost from larger

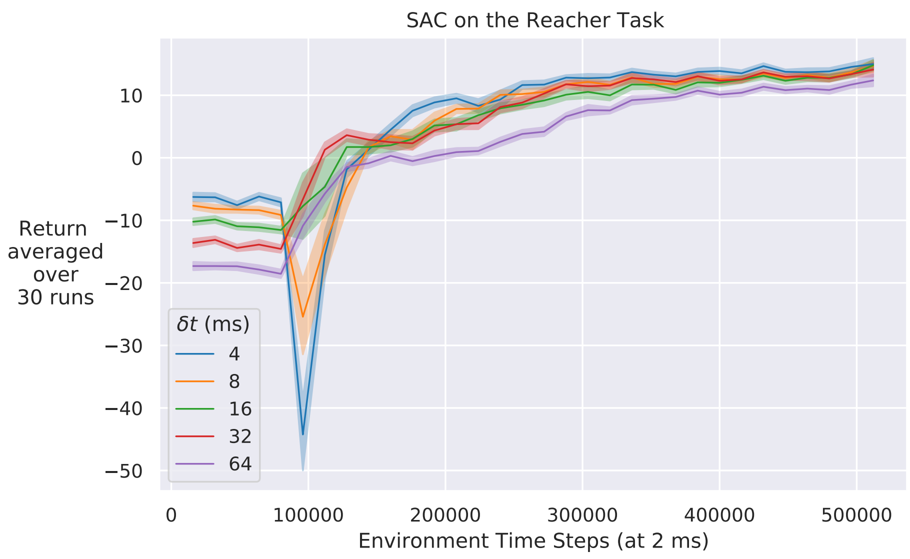

Figure 7.1: Learning curves for different δts with the baseline SAC hyper-parameters. Smaller δts suffer a sharp decline in the beginning, and the largest δt learns more slowly.

joint velocities, as the joints receive the same torque for longer durations at larger δts.

Figure 7.2 shows that exponentiating the baseline γ to the $\delta t/16$ power surprisingly makes SAC more sensitive to δt, as it boosts the performance of larger δts and hinders that of smaller ones. Tuning γ, however, can markedly improve the results of all δts. The tuned γ values are different from the baseline $\gamma = 0.99$ and shown in Table 7.1. We have demonstrated that the baseline hyper-parameter values of SAC may not be robust to different δts and simply scaling γ may not guarantee an improvement for all δts.

tuned	δt (ms)				
hyper-parameter	4	8	16	32	64
γ	0.961	0.923	0.851	0.851	0.851

Table 7.1: Tuned γ of SAC on the Reacher Task.

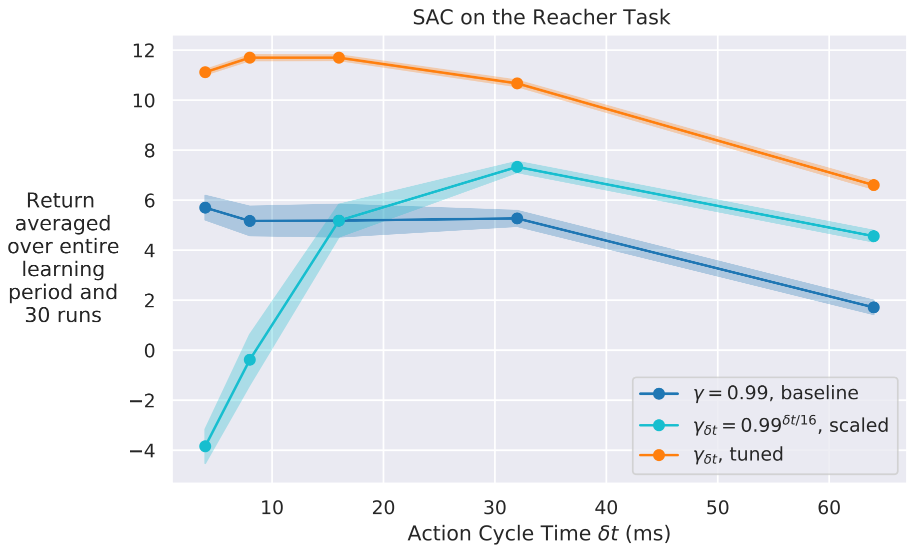

Figure 7.2: Overall average return vs. δt for different choices of γ. Scaling γ based on δt makes the performance more sensitive to δt. Performance can be thoroughly improved by tuning γ for each δt separately.

7.3 Scaling Baseline γ as a Function of δt

Since tuning γ significantly outperformed the baseline γ and its scaled variant, we performed a comprehensive experiment to better understand the relationship between γ and δt. We ran SAC with an extensive range of γ values from approximately 0.276 to 1.0 and plotted the learning curves averaged over 30 independent runs for each γ in Figure 7.3. Shaded regions indicate standard errors. All results use undiscounted returns, and other details were unchanged from the previous section.

The figure shows that, as γ is increased, the asymptotic performance for all δts marches upward, flourishes at intermediate values of γ, and thereafter proceeds to lapse into mediocrity. The magnitude of the rise and fall is more pronounced for smaller δts for which the asymptotic performance appears more sensitive to the value of γ. The learning speed seems to improve for larger δts and decline for smaller δts as γ is increased. The smaller δts start to endure a steep fall and rise of performance from the baseline $\gamma = 0.99$ onward.

To inspect the relationship between γ and δt more easily, from the corresponding learning curves we calculated the overall average return of each γ

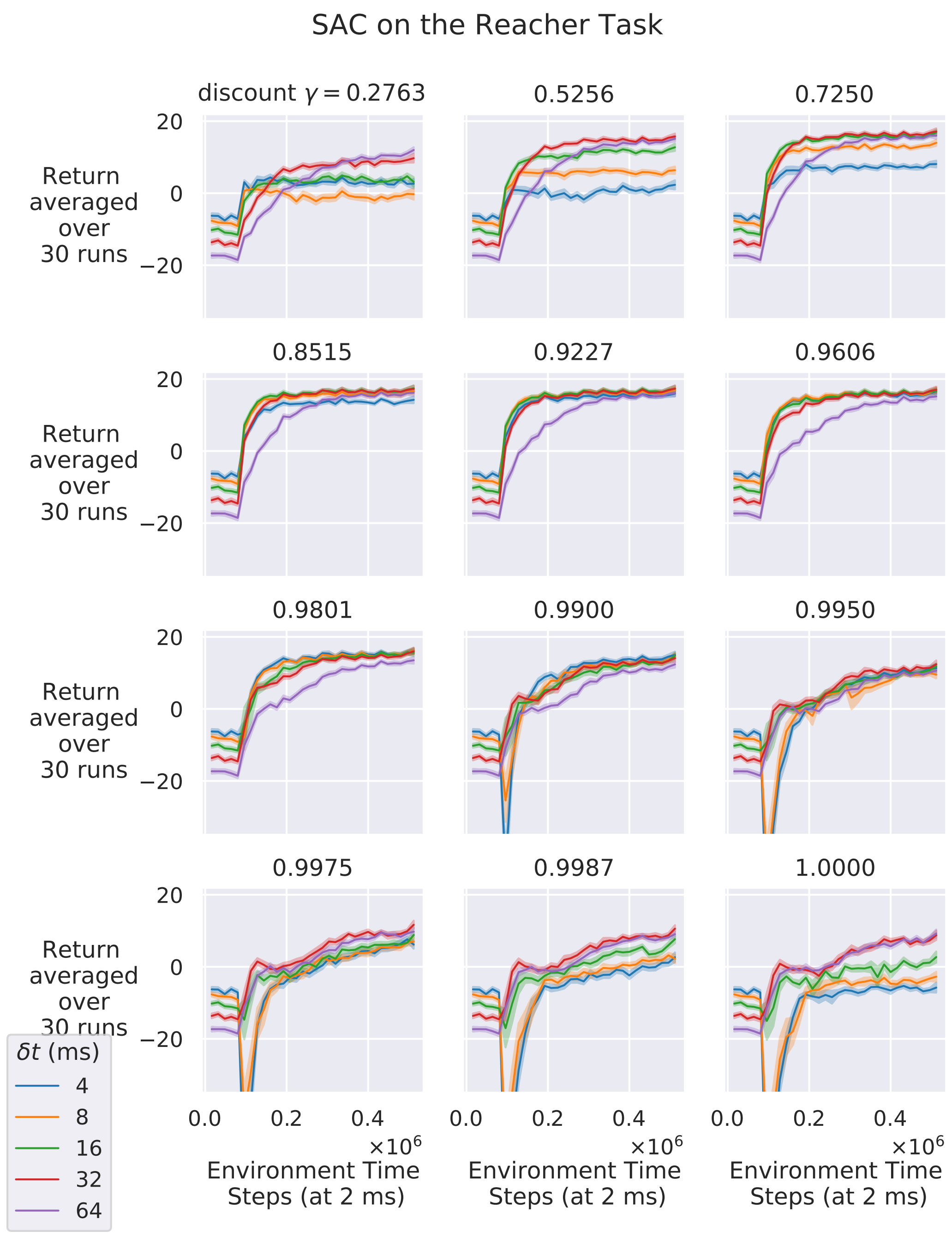

Figure 7.3: Learning curves of SAC for a sweep of γ values. Intermediate values of γ obtain better asymptotic performance and learning speed.

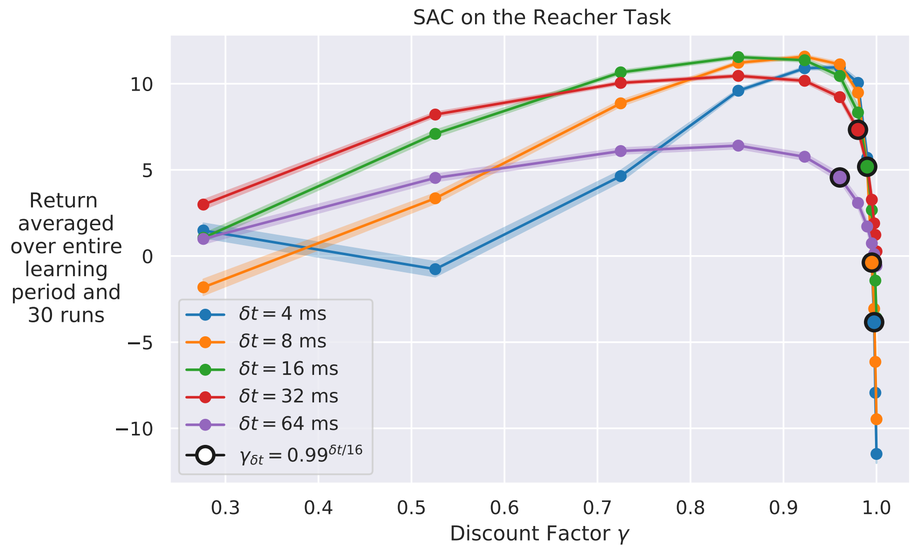

Figure 7.4: Overall average return for a sweep of γ values. Peak performance is achieved at intermediate values of γ. Curves are stretched horizontally for increasing δt, suggesting the need for scaling γ based on δt.

by averaging the return over the entire learning period and the 30 runs. Figure 7.4 depicts the results. The best performance at $\delta t = 16$ ms is reached with $\gamma \approx 0.851$, which is appreciably far from the baseline value of 0.99. The performance of all δts peak at different intermediate values of γ and steadily decline for both increasing and decreasing γs (except for the curious jump of $\delta t = 4$ ms at the smallest γ). The location of these peaks might be different for other tasks with some tasks obtaining the maximum performance with γs larger than the baseline 0.99. In this task, the baseline $\gamma = 0.99$ is larger than the best, and scaling it according to δt hurts the performance of smaller δts (shown with circle marks in Figure 7.4). If the best γ was larger than the baseline instead, scaling it would have helped smaller δts. This conflict points to the insufficiency of merely scaling γ for adapting it to different δts. In addition, as δt gets larger, the curves are consistently stretched horizontally, which hints at the need for scaling γ based on δt to make the curves more aligned.

We demonstrated in this chapter that the baseline hyper-parameter values of SAC might not be robust to different δts and simply scaling γ according to δt may not improve the robustness across all δts. Our exhaustive sweep of γ values revealed that, for all δts, the peak performance on this task was produced by a value of γ around which the performance was less sensitive to changes in γ, and that the sensitivity of performance to γ is increased as the cycle time δt is reduced.

Chapter 8

Our Proposed δt-Aware Discount Factor of SAC

We propose our approach for choosing the discount factor of the Soft Actor-Critic (SAC) algorithm in this chapter. The baseline discount factor γ of SAC is not robust to different δts and tuning it can improve the performance for all δts as shown previously. We hypothesize that the performance of our δt-aware $\gamma_{\delta t}$ is superior to that of the baseline γ at all δts. We describe two δt-aware discount factors and compare them to the baseline γ on the Reacher Task.

8.1 The δt-Aware Discount Factor

For all δts, performance of SAC is less sensitive to the discount factor around γ values that produce peak performance as revealed by the exhaustive sweep of γ values in the previous chapter. In addition, the performance becomes increasingly sensitive to γ as δt is decreased. Based on these observations, we propose two hypotheses for adapting γ to different δts. In the first, γ is initially tuned to find the best value at a baseline δt_0 and subsequently scaled based on δt. In the second, the tuned γ is kept constant for other δts. These two hypotheses are respectively expressed more formally as the new *δt-aware* *discount factor* $\gamma_{\delta t}$ for SAC as

$$\gamma_{\delta t,1} \doteq \left(\operatorname*{argmax}_{\gamma_{\delta t_0}} \hat{G}_0 \right)^{\delta t / \delta t_0} \tag{8.1}$$

$$\gamma_{\delta t,2} \doteq \left(\operatorname*{argmax}_{\gamma_{\delta t_0}} \hat{G}_0 \right), \tag{8.2}$$

with δt_0 as the baseline cycle time and $\hat{G}_0$ as the estimate of the undiscounted episodic return obtained as a sample average over the entire learning process.

8.2 Experiments

We tested the two hypotheses for the δt-aware $\gamma_{\delta t}$ on the same Reacher Task with the baseline $\delta t_0 = 16$ ms, for which $\gamma_{16} \approx 0.851$ obtained the best performance. The best performing $\gamma_{16} \approx 0.851$ was raised to the $\delta t/\delta t_0$ power for all δts in the first set of runs (tuned & scaled) and was kept constant in the second set (tuned). Both sets were independent of the runs used to determine the best γ_{16} to avoid maximization bias. The learning curves for these two hypotheses were plotted against the baseline $\gamma = 0.99$ in Figures 8.1 and 8.2 respectively. All results are averaged over 30 independent runs. To make the comparison easier, we once again calculated the average return over the entire learning period and the 30 runs for each learning curve with the results displayed in Figure 8.3. All results use undiscounted returns. Shaded regions of all plots show the standard errors.

8.3 Results and Discussion

The learning curves of both of these hypotheses in Figures 8.1 and 8.2 show significant improvement over those of the baseline $\gamma = 0.99$. Both $\gamma_{\delta t,1}$ and $\gamma_{\delta t,2}$ avoid the sharp drops that $\delta t = 4$ and $\delta t = 8$ ms experienced with the baseline γ. In addition, they both learn noticeably faster and achieve marginally better asymptotic performance. The learning curves of $\gamma_{\delta t,1}$ and $\gamma_{\delta t,2}$ bear an unremarkable resemblance to one another. The only minute difference when using $\gamma_{\delta t,2}$ seems to be a slightly improved asymptotic performance for $\delta t = 64$

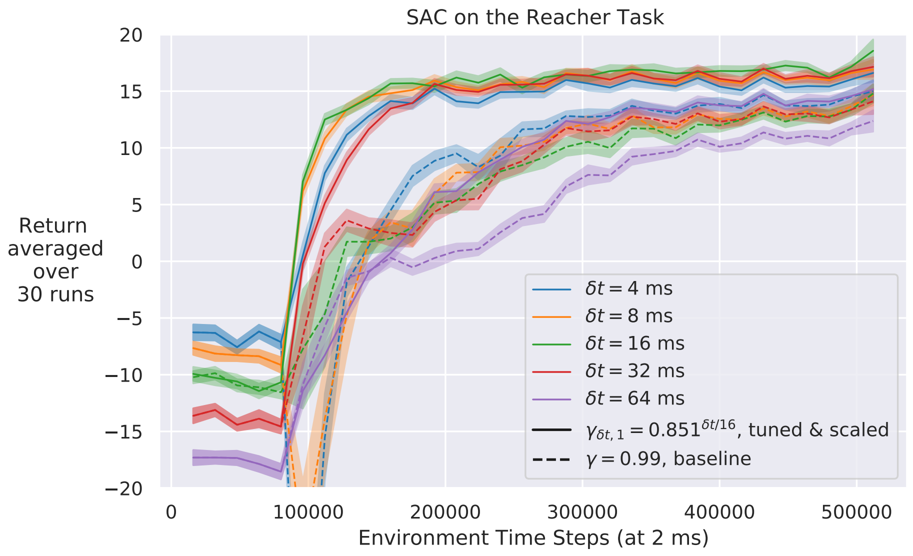

Figure 8.1: Learning curves of SAC comparing the baseline with the tuned and scaled $\gamma_{\delta t,1} = 0.851^{\delta t/16}$. All δts learn faster and avoid the sharp drops of smaller δts with the baseline γ.

ms and a slightly reduced one for $\delta t = 4$ ms. Looking at the summaries in Figure 8.3, we see that both of these hypotheses attain performance significantly superior to that of the baseline $\gamma = 0.99$. The general downward trend of performance with increasing δt could hint at the benefit of small δts over large ones on this task.

The empirical results of this chapter indicate that both hypotheses for the δt-aware discount factor of SAC perform noticeably better than the baseline γ on the Reacher Task. It is thus clear that tuning γ at a baseline δt_0 is a crucial step in adapting it to different δts. The additional scaling of the tuned $\gamma_{\delta t,2}$ based on δt benefits the performance of smaller δts and hurts that of the larger ones. Next, we validate the δt-aware discount factor on the two held-out tasks described previously.

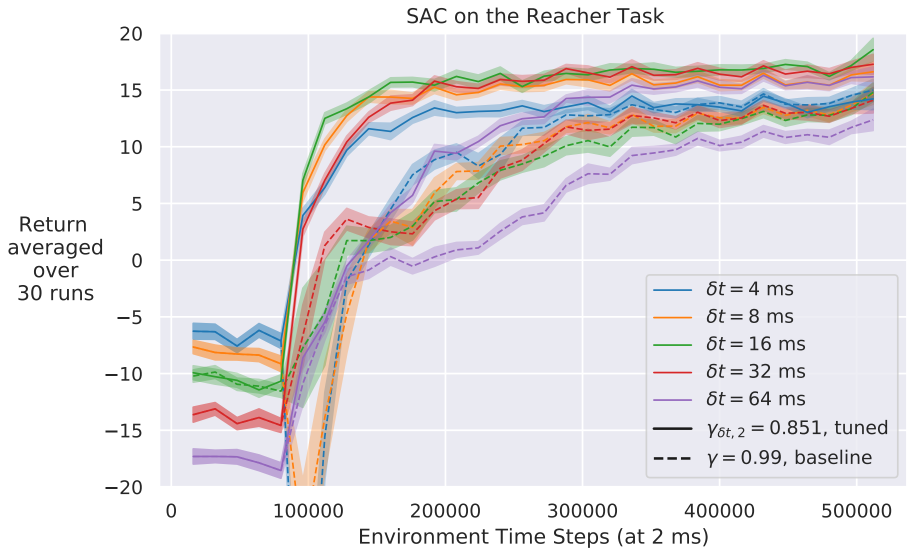

Figure 8.2: Learning curves of SAC comparing the baseline with the tuned $\gamma_{\delta t,2} = 0.851$. All δts learn faster and avoid the sharp drops of smaller δts with the baseline γ.

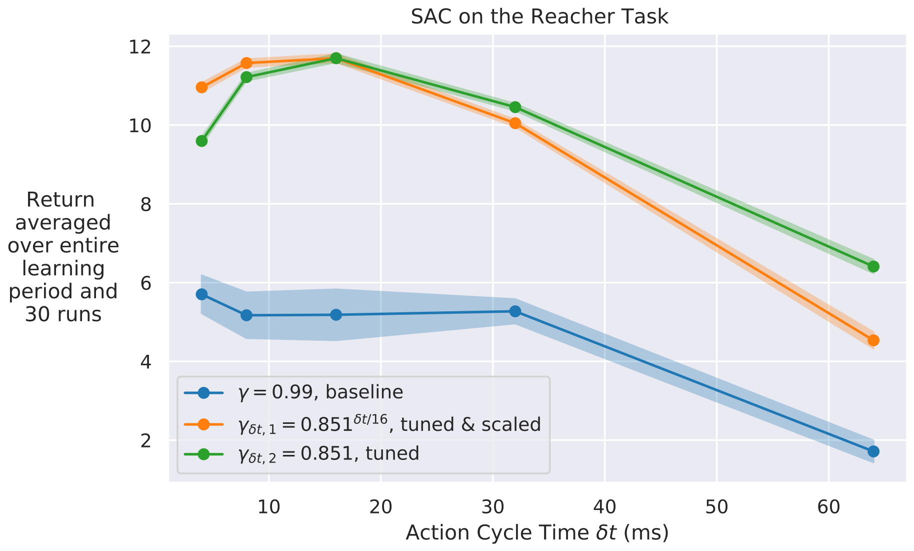

Figure 8.3: Overall average return of the two hypotheses compared with the baseline γ. Both hypotheses offer notable performance gains over the baseline, and scaling γ according to δt favors smaller δts.

Chapter 9

Validating the δt-Aware Discount Factor of SAC

In this chapter, we validate our proposed δt-aware discount factor of Soft Actor-Critic (SAC) on two previously unseen tasks. The δt-aware $\gamma_{\delta t}$ results in significant performance improvements over the baseline γ when δt changes in the original Reacher Task. We ask ourselves whether the δt-aware $\gamma_{\delta t}$ and its improvements extend beyond the Reacher Task to other environments and real-world tasks. As such, we validate the two δt-aware hypotheses by comparing them with the baseline γ on the held-out simulated and real-world robotic tasks described previously in Chapter 6.

9.1 Validation on Double Pendulum

We investigated the validity of the δt-aware discount factor $\gamma_{\delta t}$ of SAC by testing both hypotheses on the Double Pendulum Task from Section 6.1 and comparing them with the baseline $\gamma = 0.99$. Peak performance at $\delta t_0 = 16$ ms was achieved with $\gamma = 0.9987$, which was both held constant across and scaled based on different δts in two sets of experiments independent of the ones for determining the peak to avoid maximization bias. All results were averaged over 30 independent runs, each of which lasted for one million environment steps. The learning curves for the baseline $\gamma = 0.99$ were plotted in Figure 9.1, and Figures 9.2 and 9.3 compare those of the baseline γ with the δt-aware discount factors $\gamma_{\delta t,1}$ and $\gamma_{\delta t,2}$ respectively. These three figures were summarized by calculating the average return over entire learning period and the 30 runs of each learning curve and depicting them in Figure 9.4 for comparison. Shaded regions represent standard errors. All results use undiscounted returns, and other experimental details were similar to Section 6.2.

With the baseline $\gamma = 0.99$ in Figure 9.1, larger δts only experience a minor reduction in learning speed and a slight increase in asymptotic performance over the baseline $\delta t_0 = 16$ ms. The baseline $\gamma = 0.99$ particularly impairs the learning speed of smaller cycle times $\delta t = 4$ and $\delta t = 8$ ms to the extent that they do not reach their asymptote in the same time allotment as other δts. The disappointing performance of the smaller δts in this environment contrasts the outcomes from the previous environment shown in Figure 8.3 for the baseline $\gamma = 0.99$. The baseline value of γ in the SAC algorithm, therefore, may not be robust to different environments at different δts, further supporting the need for our proposed δt-aware $\gamma_{\delta t}$.

Both $\gamma_{\delta t,1}$ and $\gamma_{\delta t,2}$ profoundly revive the performance of smaller δts as seen in Figures 9.2 and 9.3. The learning speed of the baseline $\delta t_0 = 16$ ms is slightly enhanced. Larger δts were not plotted since no major change was observed for them from the baseline γ. The learning curves of both hypotheses $\gamma_{\delta t,1}$ and $\gamma_{\delta t,2}$ are especially similar to each other with no major discernible difference, except maybe for the tenuous rise in the learning speed of $\delta t = 4$ ms with $\gamma_{\delta t,1}$.

Figure 9.4 validates the δt-aware discount factor $\gamma_{\delta t}$ of SAC, as the requisite tuning of γ done by $\gamma_{\delta t,2}$ at a baseline δt_0 retrieves the majority of the lost performance at smaller δts with only marginal refinement resulting from the subsequent scaling done by $\gamma_{\delta t,1}$ based on δt.

9.2 Validation on Real-Robot Reacher

We validated the δt-aware discount factor of SAC on the Real-Robot Reacher Task from Section 6.3. The environment time step was set to 40 ms to ensure the learning updates of canonical implementations of SAC take no longer than a cycle time, and a $\gamma \approx 0.9227$ produced peak performance at the baseline $\delta t_0 = 40$ ms. Three sets of experiments at $\delta t = 120$ ms were then performed to

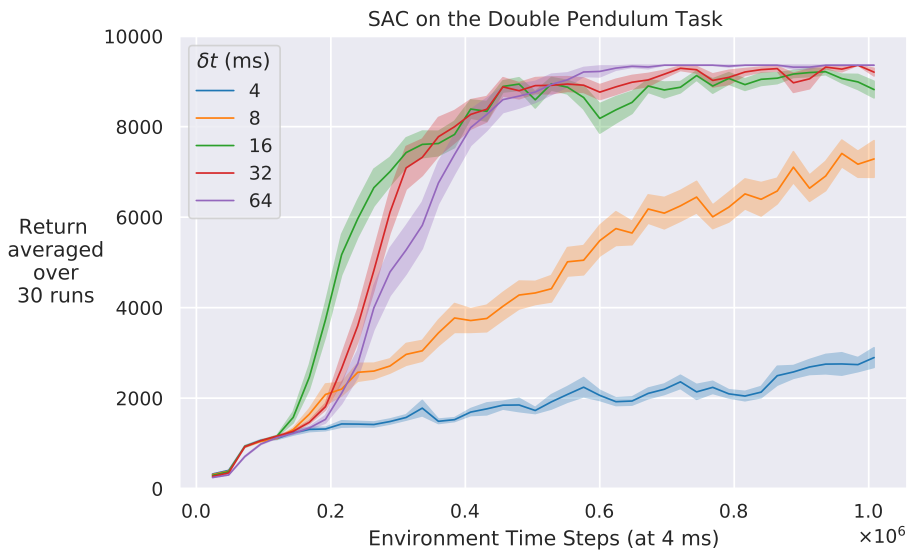

Figure 9.1: Learning curves of SAC using the baseline γ. Learning speed of smaller δts is significantly impaired, and they seemingly could not reach their asymptote in the time given.

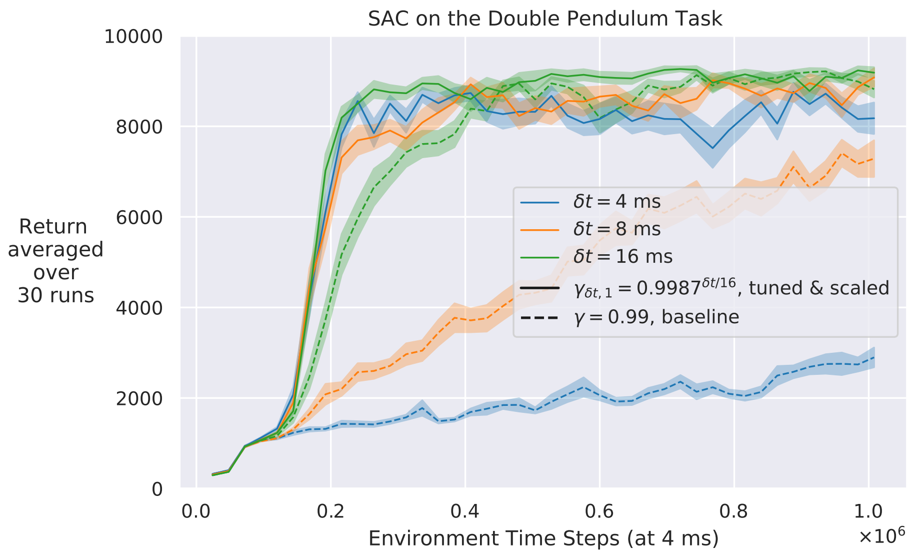

Figure 9.2: Learning curves of SAC comparing the baseline with the tuned and scaled $\gamma_{\delta t,1} = 0.9987^{\delta t/16}$. Performance of smaller δts is recovered using the tuned and scaled $\gamma_{\delta t,1}$. Larger δts were not drawn since no major change was observed compared to the baseline γ.

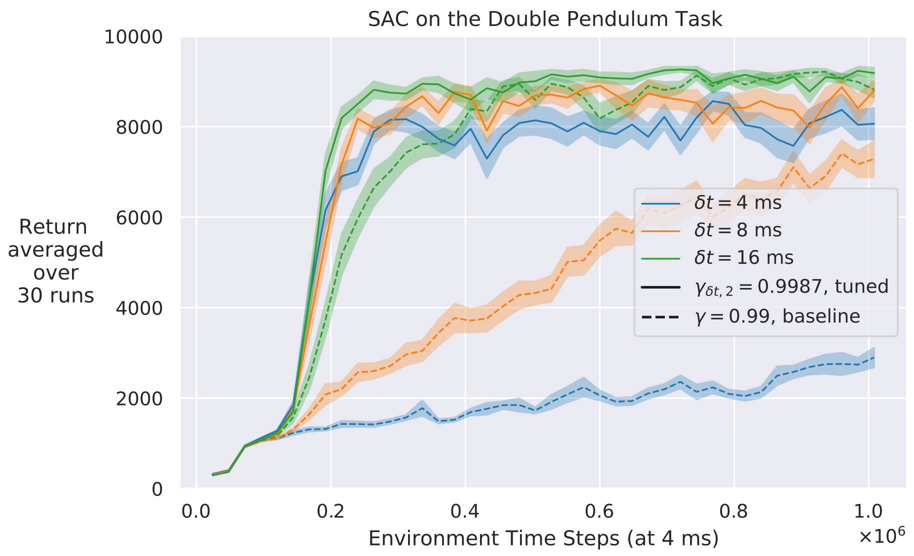

Figure 9.3: Learning curves of SAC comparing the baseline with the tuned $\gamma_{\delta t,2} = 0.9987$. Performance of smaller δts is recovered. Larger δts were not drawn since no major change was observed compared to the baseline γ.

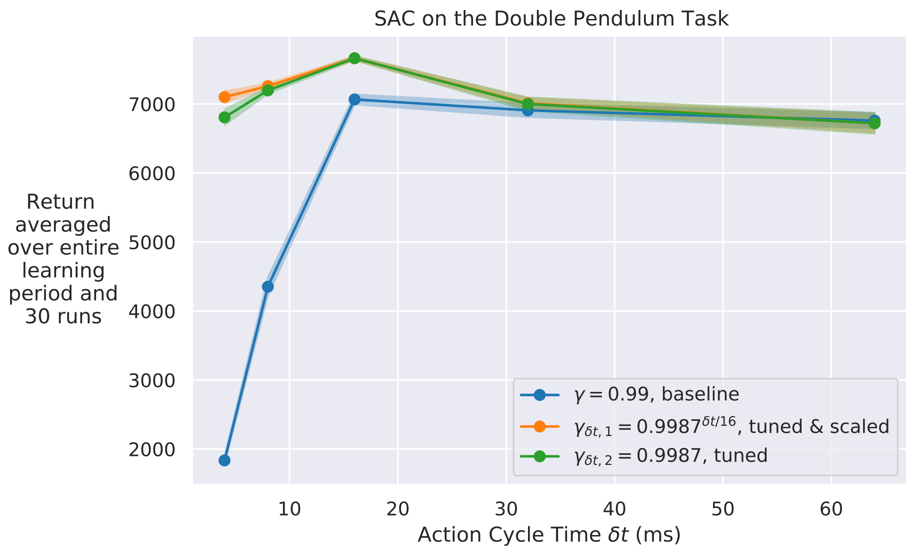

Figure 9.4: Overall average return of the two hypotheses for setting γ of SAC compared with the baseline γ. The $\gamma_{\delta t,2}$ tuned for the baseline δt_0 fully regains the performance of smaller δts. Additional scaling ($\gamma_{\delta t,1}$) causes only a marginal refinement.

examine the utility of our δt-aware $\gamma_{\delta t}$ by using the baseline $\gamma = 0.99$, the tuned $\gamma_{\delta t,2} \approx 0.9227$, and the tuned and scaled $\gamma_{\delta t,1} \approx 0.9227^{120/40} = 0.786$ each for 10 independent runs. The runs for the best-performing γ at $\delta t_0 = 40$ ms were unfortunately not repeated, and some maximization bias might be present when comparing its performance to other γs and δts. Other experimental details were unchanged from Section 6.4. The learning curves for the $\delta t_0 = 40$ and $\delta t = 120$ ms runs were drawn in Figures 9.5 and 9.6 respectively. Figure 9.7 shows the returns averaged over the 10 runs and first 50,000 environment steps or 33 real-time minutes for each set of experiments. All results use undiscounted returns. Shaded regions show standard errors.

In our SAC implementation, the learning updates happened between choosing an action and applying it to the environment. Although performing the updates after applying the action would have been preferable, this issue is orthogonal to our studies. Firstly, we used the same implementation for all of our experiments. Secondly, the place of learning updates is unimportant in simulated environments since the environment is paused between consecutive steps. Thirdly, each learning update took about 10 to 15 ms in the Real-Robot Reacher Task which is considerably lower than the baseline $\delta t_0 = 40$ ms.

Figure 9.5 shows that different values of γ perform roughly similarly at $\delta t_0 = 40$ ms, except for a small decrease in learning speed and asymptotic performance for $\gamma = 0.725$. The learning curves of Figure 9.6 reveal the asymptotic performance as the only source of difference between the three different γ values at $\delta t = 120$ ms. This signifies the importance of choosing the optimal γ for each δt since a suboptimal γ might not perform as well as an optimal one even after extended learning.

These learning curves can be compared more readily in Figure 9.7. The baseline $\gamma = 0.99$ reduces the performance when δt is changed from 40 to 120 ms. Using the tuned $\gamma_{\delta t,2} \approx 0.9227$ provides a modest recovery, and the additional scaling based on δt regains a performance almost similar to the peak at $\delta t_0 = 40$ ms. The results once again indicate that tuning γ for a baseline δt_0 is an essential step in adapting γ to different δts. The high performance achieved at $\delta t = 120$ ms with an atypical $\gamma_{\delta t,1} \approx 0.9227^{120/40} =$

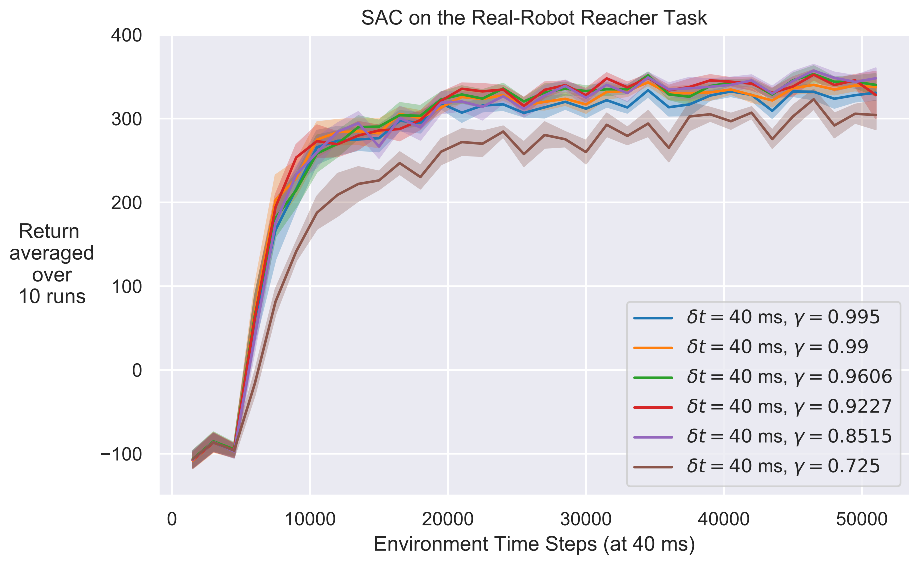

Figure 9.5: Learning curves of SAC on the Real-Robot Reacher Task for the sweep of γ values at $\delta t_0 = 40$ ms. The smallest $\gamma = 0.725$ performs slightly worse than other γs. Other γs are roughly similar in learning speed and asymptotic performance.

0.786 demonstrates the necessity of having guidelines such as our δt-aware $\gamma_{\delta t}$ to avoid the poor performance of the baseline $\gamma = 0.99$ when changing δt.

Overall, the δt-aware discount factor of SAC performs significantly better than the baseline γ on both simulated and real-world held-out tasks. The discount factor γ should be tuned to a specific δt for it to perform well when transferred to a different δt on the same task. Scaling the best-performing γ with respect to δt may improve the performance of smaller δts marginally. The δt-aware discount factor may reduce the need for costly and time-consuming hyper-parameter tuning in the real world when the δt of a task changes.

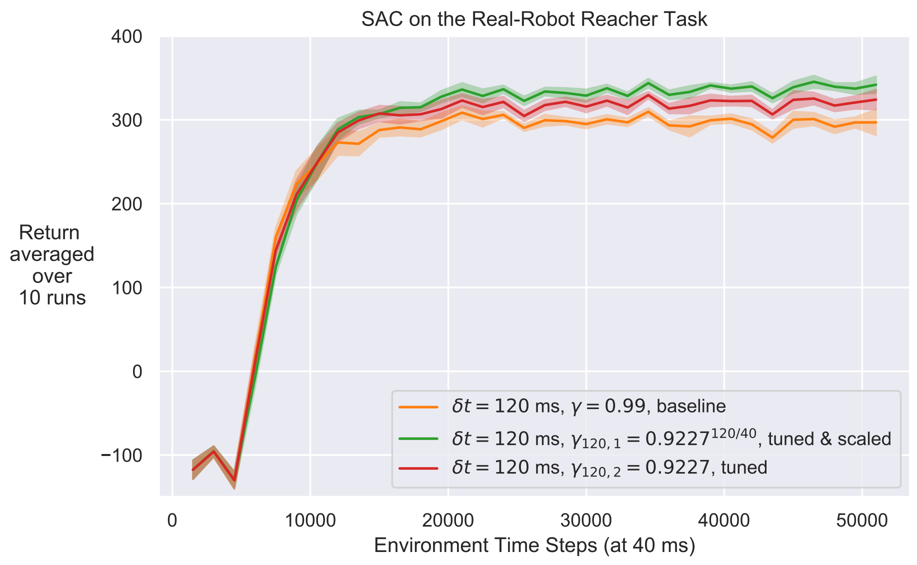

Figure 9.6: Learning curves of SAC on the Real-Robot Reacher Task for $\delta t =$ 120 ms. The asymptotic performance of the baseline $\gamma = 0.99$ cannot reach that of the atypical δt-aware $\gamma_{\delta t,1} \approx 0.786$ even after extended learning.

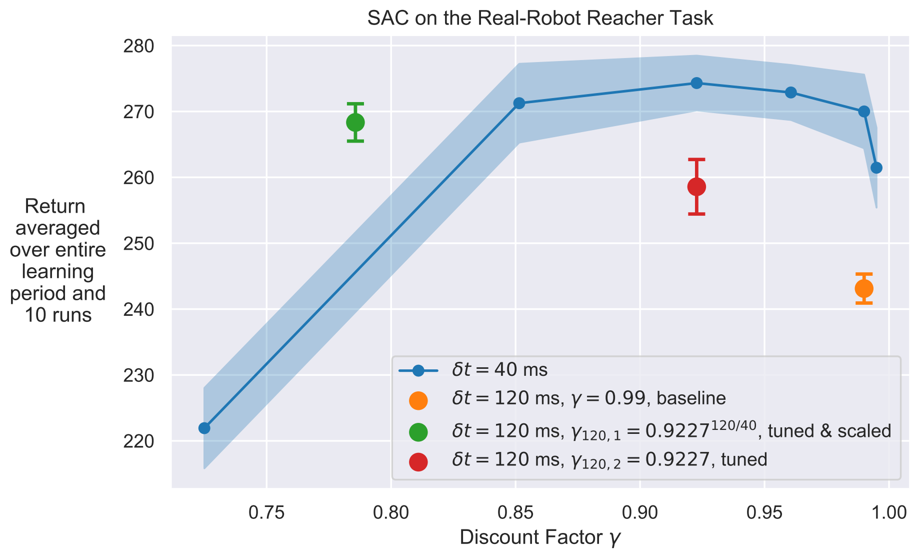

Figure 9.7: Overall average return of SAC comparing the baseline γ with our two hypotheses on the Real-Robot Reacher Task. Tuning $\gamma_{\delta t,2}$ for the baseline δt_0 and scaling it based on δt to get $\gamma_{\delta t,1}$ almost fully recovers the poor performance of the baseline γ on the changed δt.

Chapter 10

Conclusion

Our study of different time discretizations and their effects on two different policy gradient algorithms comes to an end in this chapter. Algorithm hyper-parameters may need to be tuned when the action-cycle time δt changes in a given task. This tuning can be time-consuming and costly for real-world robots, given their slow real-time rate of experience collection and proneness to wear and tear and breakdown. Having hyper-parameters that are robust to different δts can allow hyper-parameter values tuned to a particular δt to be transferred to different δts on the same task.

In this thesis, we focused on the hyper-parameter values of the two popular policy gradient algorithms Proximal Policy Optimization and Soft Actor-Critic. We demonstrated that the baseline hyper-parameter values of these algorithms may not be robust to different δts and took a step toward improving their robustness. For both PPO and SAC, we investigated the relationship between some of their hyper-parameters and their performance at different δts on the Reacher Task. Based on the observed relationships, we presented our δt-aware hyper-parameters that adjust based on δt and empirically showed that they make the performance robust to different δts. The δt-aware hyper-parameters adapt four different hyper-parameters in PPO and only the discount factor in SAC.

The PPO algorithm collects a new batch of data with size b for each iteration of updates. Changing δt affects how quickly or slowly in real-time this batch is collected and how much real-time information it represents. It appears that PPO requires a minimum amount of real-time information in each batch to perform well. In addition, the mini-batch size m determines the number of gradient steps taken for each batch, and reducing δt means that the same number of gradient steps have to be taken in less time, potentially making smaller δts more compute-intensive. Reducing δt results in the discount factor γ and the trace-decay parameter λ decaying faster in real-time, which seems to be desirable for smaller δts in PPO. For SAC, the range of well-performing values for the discount factor γ appears to get narrower as δt is decreased, and performance seems less sensitive to the discount factor around γ values that produce peak performance.

We validated the δt-aware hyper-parameters on the two held-out tasks Double Pendulum and Real-Robot Reacher. These hyper-parameters are more robust to different δts on these tasks and can enable the transfer of hyper-parameter values tuned to a specific δt on these tasks to a different δt.

Comparing the results of different environments together reveals that the baseline hyper-parameter values of PPO and SAC may not be robust to different environments at δts other than the baseline δt for which the hyper-parameters are tuned. This shows a second weakness of the baseline values in addition to their lack of robustness to different δts in a single environment. Overall, our extensive experiments amounted to 75 real-time hours or 10 million time steps of real-world robot learning, 20,000 simulation hours or 7 billion time steps of simulation with PPO, and 1000 simulation hours or 350 million time steps of simulation with SAC.

We introduced three different tasks that were modified to support different δts and explained how the agent-environment interaction loop of any algorithm can be altered to enable a fair comparison of performance among different δts. We make our implementations publicly available to facilitate further studies on δt, perhaps on future algorithms and more environments. We hope that our contributions pave the way for future studies to consider δt as a tunable hyper-parameter and experiment with different values of δt.

References

Akkaya, I., Andrychowicz, M., Chociej, M., Litwin, M., McGrew, B., Petron, A., Paino, A., Plappert, M., Powell, G., Ribas, R., Schneider, J., Tezak, N., Tworek, J., Welinder, P., Weng, L., Yuan, Q., Zaremba, W., Zhang, L. (2019). Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.0 7113.*

Baird, L. C. (1994). Reinforcement learning in continuous time: Advantage updating. In *Proceedings of 1994 IEEE International Conference on Neural Networks.*

Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P. P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., Pinto, H. P. O., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F., Zhang, S. (2019). Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680.*

Bhasin, S., Kamalapurkar, R., Johnson, M., Vamvoudakis, K. G., Lewis, F. L., Dixon, W. E. (2013). A novel actor–critic–identifier architecture for approximate optimal control of uncertain nonlinear systems. *Automatica 49* (1):82–92.

Chen, B., Xu, M., Li, L., Zhao, D. (2021). Delay-aware model-based reinforcement learning for continuous control. *Neurocomputing 450*:119–128.

Coumans, E., Bai, Y. (2016). PyBullet, a python module for physics simulation for games, robotics and machine learning. URL `http://pybullet.org`

Doya, K. (1996). Temporal difference learning in continuous time and space. In *Advances in Neural Information Processing Systems.*

Doya, K. (2000). Reinforcement learning in continuous time and space. *Neural Computation 12* (1):219–245.

Du, J., Futoma, J., Doshi-Velez, F. (2020). Model-based reinforcement learning for semi-Markov decision processes with neural ODEs. In *Advances in Neural Information Processing Systems.*

Dulac-Arnold, G., Levine, N., Mankowitz, D. J., Li, J., Paduraru, C., Gowal, S., Hester, T. (2021). Challenges of real-world reinforcement learning: Definitions, benchmarks and analysis. *Machine Learning.* Advance online publication. `https://doi.org/10.1007/s10994-021-05961-4`.

Firoiu, V., Ju, T., Tenenbaum, J. (2018). At human speed: Deep reinforcement learning with action delay. *arXiv preprint arXiv:1810.07286.*

Fujimoto, S., Hoof, H., Meger, D. (2018). Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning.*

Haarnoja, T., Zhou, A., Abbeel, P., Levine, S. (2018a). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning.*

Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., Levine, S. (2018b). Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905.*

Haarnoja, T., Ha, S., Zhou, A., Tan, J., Tucker, G., Levine, S. (2018c). Learning to walk via deep reinforcement learning. In *Robotics: Science and Systems XV.*

Harmon, M. E., Baird, L. C., Klopf, A. H. (1995). Advantage updating applied to a differential game. In *Advances in Neural Information Processing Systems.*

Johnson, M., Bhasin, S., Dixon, W. E. (2011). Nonlinear two-player zero-sum game approximate solution using a policy iteration algorithm. In *2011 50th IEEE Conference on Decision and Control and European Control Conference.*

Kim, J., Yang, I. (2020). Hamilton-Jacobi-Bellman equations for Q-learning in continuous time. In *Proceedings of the 2nd Conference on Learning for Dynamics and Control.*

Kingma, D. P., Ba, J. (2014). Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations.*

Lan, Q., Mahmood, A. R. (2021). Model-free policy learning with reward gradients. *arXiv preprint arXiv:2103.05147.*

Lee, J., Sutton, R. S. (2021). Policy iterations for reinforcement learning problems in continuous time and space—fundamental theory and methods. *Automatica 126*, Article 109421.

Li, H., Liu, D., Wang, D. (2014). Integral reinforcement learning for linear continuous-time zero-sum games with completely unknown dynamics. *IEEE Transactions on Automation Science and Engineering 11* (3):706–714.

Mahmood, A. R., Korenkevych, D., Komer, B. J., Bergstra, J. (2018a). Setting up a reinforcement learning task with a real-world robot. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems.*

Mahmood, A. R., Korenkevych, D., Vasan, G., Ma, W., Bergstra, J. (2018b). Benchmarking reinforcement learning algorithms on real-world robots. In *Proceedings of the 2nd Annual Conference on Robot Learning.*

Marbach, P., Tsitsiklis, J. N. (2001). Simulation-based optimization of Markov reward processes. *IEEE Transactions on Automatic Control 46* (2):191–209.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature 518* (7540):529–533.

Modares, H., Lewis, F. L. (2014). Linear quadratic tracking control of partially-unknown continuous-time systems using reinforcement learning. *IEEE Transactions on Automatic Control 59* (11):3051–3056.

Munos, R. (2006). Policy gradient in continuous time. *Journal of Machine Learning Research 7* (27):771–791.

Munos, R., Bourgine, P. (1998). Reinforcement learning for continuous stochastic control problems. In *Advances in Neural Information Processing Systems.*

Nota, C., Thomas, P. S. (2020). Is the policy gradient a gradient? In *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems.*

Pardo, F., Tavakoli, A., Levdik, V., Kormushev, P. (2018). Time limits in reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning.*

Ramstedt, S., Pal, C. (2019). Real-time reinforcement learning. In *Advances in Neural Information Processing Systems.*

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347.*

Singh, S. P., Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine learning 22* (1-3):123–158.

Sutton, R. S., Barto, A. G. (2018). *Reinforcement Learning: An Introduction.* MIT Press.

Sutton, R. S., McAllester, D. A., Singh, S. P., Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems.*

Tallec, C., Blier, L., Ollivier, Y. (2019). Making deep Q-learning methods robust to time discretization. In *Proceedings of the 36th International Conference on Machine Learning.*

Tan, J., Zhang, T., Coumans, E., Iscen, A., Bai, Y., Hafner, D., Bohez, S., Vanhoucke, V. (2018). Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332.*

Travnik, J. B., Mathewson, K. W., Sutton, R. S., Pilarski, P. M. (2018). Reactive reinforcement learning in asynchronous environments. *Frontiers in Robotics and AI 5*:79.

Vamvoudakis, K. G., Lewis, F. L. (2010). Online actor–critic algorithm to solve the continuous-time infinite horizon optimal control problem. *Automatica 46* (5):878–888.

van Hasselt, H. (2010). Double Q-learning. In *Advances in Neural Information Processing Systems.*

Williams, R. J. (1987). Reinforcement-learning connectionist systems. Technical Report NU-CCS-87-3. College of Computer Science, Northeastern University, Boston.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning 8* (3-4):229–256.

Xiao, T., Jang, E., Kalashnikov, D., Levine, S., Ibarz, J., Hausman, K., Herzog, A. (2020). Thinking while moving: Deep reinforcement learning with concurrent control. In *8th International Conference on Learning Representations.*

Zambrano, D., Roelfsema, P. R., Bohte, S. M. (2015). Continuous-time on-policy neural reinforcement learning of working memory tasks. In *2015 International Joint Conference on Neural Networks.*