# Asynchronous Reinforcement Learning for Real-Time Control of Physical Robots

by

Yufeng Yuan

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

Master of Science

Department of Computing Science

University of Alberta

© Yufeng Yuan, 2021

# Abstract

An oft-ignored challenge of real-world reinforcement learning is that, unlike standard simulated environments, the real world does not pause when agents make learning updates. As standard simulated environments do not address this real-time aspect of learning, most available implementations of deep reinforcement learning algorithms process environment interactions and learning updates sequentially. Consequently, when such implementations are deployed in the real world, they may not act responsively and learn efficiently. Asynchronous learning has been proposed to solve this issue, but no systematic comparison between sequential and asynchronous reinforcement learning was conducted using real-world environments. In this thesis, we set up two vision-based tasks with a robotic arm, implement an asynchronous learning system that extends a previous architecture, and compare sequential and asynchronous reinforcement learning across different action cycle times, sensory data dimensions, and mini-batch sizes. Our experiments show that when the time cost of learning updates increases, the action cycle time in sequential implementation could grow excessively long, while the asynchronous implementation can always maintain a fixed and appropriate action cycle time. Consequently, when learning updates are expensive, the performance of sequential learning diminishes and is outperformed by a substantial margin by asynchronous learning. Our system learns in real-time to reach and track visual targets from pixels within two hours of experience and does so directly using real robots, learning completely from scratch.

# Preface

Results in this thesis, including the environment setup and different learning architectures, were submitted to CORL 2021 conference. This submission was coauthored with my supervisor Prof. Rupam Mahmood.

# Acknowledgements

I am greatly grateful to my supervisor Prof. Rupam Mahmood. His meticulous guidance and constant encouragement have been indispensable in my master's career. He persists in achieving long-term goals and sticks to rigorous research standards, which I benefit tremendously from.

I am thankful for all the amazing people from our Robot LAIR group and Reinforcement Learning and Artificial Intelligence Lab. I also thank Jun Luo, Daniel Graves, and Craig Sherstan from Huawei Noah's Ark Lab for their help and support. I am also thankful to Kindred Inc. for their donation of the UR5 robotic arm.

I thank the University of Alberta, Alberta Machine Intelligence Institute (Amii), the Canada CIFAR AI Chairs Program, and Huawei Noah's Ark Lab for funding this program.

# Contents

**1 Introduction** **1**

1.1 Challenge in Real-time Reinforcement Learning . . . . . . . . 2

1.2 Asynchronous Reinforcement Learning . . . . . . . . . . . . . 3

1.3 Related Works . . . . . . . . . . . . . . . . . . . . . . . . . . 4

1.4 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . 5

**2 Background** **7**

2.1 Problem Setup . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.2 Convolutional Neural Networks . . . . . . . . . . . . . . . . . 9

2.3 The Soft Actor-Critic Algorithm . . . . . . . . . . . . . . . . . 12

**3 Proposed Vision-based Control Environment** **14**

3.1 The Physical Setup . . . . . . . . . . . . . . . . . . . . . . . . 14

3.2 Environment Interactions Implementation . . . . . . . . . . . 17

3.3 Environment Specifications . . . . . . . . . . . . . . . . . . . . 19

**4 Asynchronous Learning Architecture** **21**

4.1 The Soft Actor-Critic Implementation . . . . . . . . . . . . . . 21

4.2 The Sequential Learning Architecture . . . . . . . . . . . . . . 23

4.3 The Existing Asynchronous Learning Architecture . . . . . . . 24

4.4 Our Proposed Asynchronous Learning Architecture . . . . . . 26

4.5 Computational Comparison of the Three Architectures . . . . 27

**5 Experimental Setup** **30**

5.1 Three Experimental Settings . . . . . . . . . . . . . . . . . . . 30

5.2 Choosing Action-Cycle Time . . . . . . . . . . . . . . . . . . . 31

5.3 Experiment Methodology . . . . . . . . . . . . . . . . . . . . . . . . 32

**6 Experimental Results with The Sequential Learning Architecture 34**

6.1 Training Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

6.2 Evaluation Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

6.3 Results Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

**7 Experimental Results with The Existing Asynchronous Learning Architecture 37**

7.1 Training Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

7.2 Evaluation Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

7.3 Results Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

**8 Experimental Results with Our Proposed Asynchronous Learning Architecture 40**

8.1 Training Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

8.2 Evaluation Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

8.3 Results Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

8.4 Learned Behavior . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

8.5 Learned Representation . . . . . . . . . . . . . . . . . . . . . . . . 43

**9 Conclusion 44**

**References 46**

# List of Tables

5.1 Summary of three experimental settings. . . . . . . . . . . . . . 31

5.2 The computation time of each component in *Seq-SAC* measured in our three experimental settings. . . . . . . . . . . . . . . . . 32

5.3 SAC Hyperparameters . . . . . . . . . . . . . . . . . . . . . . . . 33

# List of Figures


2.1 Markov decision process. . . . . . . . . . . . . . . . . . . . . . . 8
2.2 A typical convolution neural network model with both convolution layers and fully-connected layers. . . . . . . . . . . . . . 10
2.3 Random cropping augmentation applied to image observations. 11

3.1 The physical setup of the environment, including the robotic arm, the wrist-mounted camera, and the monitor. . . . . . . 15
3.2 The 160 × 90 image captured by the wrist-mounted camera, where the red dot is the target. . . . . . . . . . . . . . . . . . . 16
3.3 The environment architecture in this work. . . . . . . . . . . 17
3.4 The agent-environment interaction cycle (Sutton et al. 2018). 19

4.1 The neural network architecture used in the experiments. The solid arrows show the directions of the forward pass, and the dashed arrows show the directions of the backward pass. . . . 22
4.2 Overview of the sequential learning architecture. . . . . . . . 23
4.3 Overview of the existing asynchronous learning architecture. . 25
4.4 Overview of our extended asynchronous learning architecture. 27
4.5 The computational flow over time of the three versions of SAC. For plotting purposes, the relative length of each block may not reflect the relative computation time. . . . . . . . . . . . . . 29

6.1 The learning curves of *Seq-SAC* . . . . . . . . . . . . . . . . . 34
6.2 The overall performance of *Seq-SAC* . . . . . . . . . . . . . . 35
6.3 The evaluation performance of *Seq-SAC* . . . . . . . . . . . . 35

7.1 The learning curves of *Seq-SAC* and *Async-SAC-1*. . . . . . . 37

7.2 The overall performance of *Seq-SAC* and *Async-SAC-1* . . . 38
7.3 The evaluation performance of *Seq-SAC* and *Async-SAC-1* . 38
8.1 The learning curves of *Async-SAC-2*, *Async-SAC-1*, and *Seq-SAC* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
8.2 The overall performance of *Async-SAC-2*, *Async-SAC-1*, and *Seq-SAC* . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
8.3 The evaluation performance of *Async-SAC-2*, *Async-SAC-1*, and *Seq-SAC* . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
8.4 Learned behaviors in *Reaching* (first row) and *Tracking* (second row). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
8.5 Coordinates captured by the spatial softmax layer. . . . . . . 43

# Chapter 1

# Introduction

The utilization of robots can facilitate the automation of production and improve the efficiency of society. However, robots based on classical control methods face difficulty when deployed in a dynamic environment. One approach to enable autonomous adaptation in robots is through deep RL, which uses neural networks as function approximators for reinforcement learning. Unfortunately, the potential of deep reinforcement learning has been mostly demonstrated in simulated robotic control tasks and rarely in real-world applications.

In simulated environments, such as DeepMind Control Suite (Tassa et al. 2020), OpenAI Gym (Brockman et al. 2016), and Pybullet (Coumans et al. 2016–2021), deep reinforcement learning has made remarkable progress in complex robotic control tasks (e.g., Schulman et al. 2015, Duan et al. 2016, Schulman et al. 2017, Haarnoja et al. 2018a). However, in real-world applications, the current progress is far behind the above results. Such a gap between simulated results and real-world applications is usually attributed to two factors. First, real-world environments require more physical setup, hardware tuning, and potentially expensive devices, such as a real robotic arm. Second, the real-world environments impose extra challenges to reinforcement learning, such as the slow data collection, system delay from sensory inputs, high-dimensional state space and action space, rigorous environment constraints, and partial observability (Dulac-Arnold et al. 2020). However, one oft-ignored implicit consequence of the imbalanced usage of simulated environments and real-world

environments is that most algorithms and implementations are only tested in simulated environments. When they are deployed in real-world environments, in which the agents need to learn and act in real-time, it is unclear whether they can maintain the performance as they do in simulated environments.

In this thesis, we explore the challenge of real-time reinforcement learning, investigate how order of computations affects the performance of real-time systems, evaluate asynchronous reinforcement learning as one approach to address some of the challenges, and provide recommendations for improving upon existing asynchronous learning architecture. All the results are obtained using real-world, vision-based robotic tasks.

## 1.1 Challenge in Real-time Reinforcement Learning

Besides the challenges mentioned for real-world reinforcement learning, one additional challenge that is often ignored is the sequential computation of environment interactions and learning updates in most available open-source implementations. Under this sequential computation, to sample an action at time step $t$, the agent has to wait until the learning update before time step $t$ finishes; and to make a learning update at time step $t$, the agent also has to wait until the agent-environment interactions at time step $t$ finishes. In this case, the minimal time interval between two consecutive actions is lower-bounded by the time cost of learning updates. Such computational arrangement, which is simple to implement and debug, is appropriate in simulated environments because the simulated environments can be internally paused while agents make learning updates. Likewise, in standard simulated environments, no matter how long the learning updates take, the subsequent state or observation the agent receives will always be the same. However, in the real world, time marches on regardless of the learning updates, and the environment will keep executing the previous action command before the learning update finishes. Because the control frequency of a robotic system is usually on the order of milliseconds, learning updates could take much longer than that. In this case,

the sequential computation could delay the execution of the next action when learning updates are in progress, potentially prolonging the effective length of the time step or *action cycle time*, reducing the amount of observations and gradient updates as well as the responsiveness of the agent. In practice, this issue can be considerably exacerbated when high-dimensional data, such as images, are combined with algorithms with replay buffers that perform expensive per-step updates such as Soft Actor-Critic (SAC, Haarnoja et al. 2018a or Deep Deterministic Policy Gradient (DDPG, Lillicrap et al. 2016). However, those replay buffer-based algorithms are, so far, most ideal for real-world reinforcement learning, as the utilization of replay buffer enables efficient data reuse, which copes with one of the crucial challenges, the very limited number of samples, in real-world reinforcement learning.

## 1.2 Asynchronous Reinforcement Learning

When applying sequential implementations of deep RL algorithms in the real world, the potential problem is caused by the combination of sequentially arranged computations and expensive learning updates. One approach to solving this problem is to reduce the actual time cost of the learning updates, such as using inexpensive incremental learning updates (A. Mahmood 2017). If the actual time cost of learning updates can be kept similar to the control frequency of robotic systems, it will have no negative effect on the learning agent. The other approach is to rearrange the computations by decoupling environment interactions and learning updates so that they both can proceed at their own pace without interfering with each other. This can be achieved with off-policy algorithms and running environment interactions and learning updates asynchronously on different threads or processes. Such asynchronous systems, usually referred to as asynchronous reinforcement learning or distributed reinforcement learning, have been proposed and used in large-scale experiments in simulated evironments (e.g., Nair et al. 2015, Espeholt et al. 2018, Barth-Maron et al. 2018), in the form of multiple actors interacting with multiple instances of the environment to collect data. However, those systems

may not be readily applicable to real-world robotic control tasks. They usually assume a large pool of parallel instances of the environment to accelerate data collection. At the same time, the number of robots which can be used for experiments is usually quite limited in practice. For real-world robotic control tasks, asynchronous learning systems have also been utilized previously, such as (e.g., Gu et al. 2017, Haarnoja et al. 2018c, Kalashnikov et al. 2018), to maintain appropriate action cycle time for the robotic system and accelerate data collection. However, no systematic study of the benefits of asynchronous learning over sequential learning has been conducted on real-world robotic tasks.

## 1.3 Related Works

In this thesis, we systematically compare asynchronous reinforcement learning and sequential reinforcement learning in real-world vision-based control tasks. Previous works that are most relevant to this thesis are reinforcement learning from images and real-world reinforcement learning.

Reinforcement learning from images (Mnih et al. 2013) usually needs a prohibitively large amount of data due to the relatively sparse reward signal, high-dimensional data, and partial-observability. To alleviate this issue, one common approach is to add auxiliary tasks, such as self-supervised prediction (Jaderberg et al. 2017), auto-encoder reconstruction (Yarats et al. 2019), and contrastive learning (Stooke et al. 2021) to provide more training signals to visual representations. A complimentary but highly effective approach is to add random augmentations to image input, such as random cropping proposed in Yarats et al. (2021) and Laskin et al. (2020), to regularize representations by learning an augmentation-invariant critic. However, most of the works consider simulated environments with a stationary overhead camera, and in contrast, our setting has the camera mounted at the end of a physical robotic arm.

Reinforcement learning has also been applied to a wide variety of real-world tasks, such as motor skills (Lange et al. 2012), grasping objects (Kalashnikov et

al. 2018), in-hand object manipulation (Andrychowicz et al. 2020), door opening (Gu et al. 2017), and locomotion (Haarnoja et al. 2018c). To successfully solve these tasks, different methods have been proposed to overcome the real-world challenges, such as ensuring critical constraints are never violated (Dalal et al. 2018), removing manual reset by learning a resetting policy (Eysenbach et al. 2018), removing hand-engineered reward functions (Zhu et al. 2020), utilizing prior knowledge in simulations (Peng et al. 2018), and using auxiliary tasks (Schwab et al. 2019).

Although algorithms such as reactive SARSA (Travnik et al. 2018) already utilize asynchronous learning to reduce the time it takes for an agent to react to observation, a careful study of the difference between sequential and asynchronous learning is still missing, and few of these works make their asynchronous implementation publicly available.

## 1.4 Contributions

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

- We implement two vision-based tasks with a physical robotic arm that can serve as a benchmarking environment for real-world vision-based control. The two tasks are based on the SenseAct framework (A. R. Mahmood et al. 2018a), consisting of a UR5 robotic arm performing reaching and tracking behaviors, a monitor displaying the target, and a wrist-mounted RGB camera capturing images. The first task with a stationary target can be solved by a coarse control policy, while the second task with a constantly moving target requires a finer control policy.

- We propose an asynchronous learning architecture for efficient learning from images, which extends the existing architecture (Haarnoja et al. 2018c) that runs environment interactions and learning updates in parallel. The existing architectures can be sufficient for low-dimensional data but may not be efficient enough when learning from high-resolution images. Thus, we extend this architecture by separating replay buffer sampling and gradient updates inside the learning update process into two parallel processes.

- We conduct systematic experiments to compare sequential learning and the two variants of asynchronous learning systems across different action cycle times, sensory data dimensions, and mini-batch sizes and give analysis to the empirical results obtained.
- We make our implementations of the tasks and the learning system publicly available to enable reproducibility and accelerate further advancement of real-world and real-time robot learning from images. Our implementation can be found in https://github.com/YufengYuan/ur5_async_rl.

# Chapter 2

# Background

This chapter presents the problem setup, the model architecture, and the specific algorithm we use for our implementation and experiments. We investigate the benefits and limitations of asynchronous reinforcement learning by formulating the problem as a finite-horizon Markov decision process (MDP). As part of the observation space is image input, we also briefly discuss the convolutional neural network, which is the most commonly used model in processing visual information. Lastly, we discuss the Soft Actor-Critic algorithm, which is the algorithm we use throughout this thesis.

## 2.1 Problem Setup

In this section, we introduce our notations and the mathematical formalism of reinforcement learning, following Kaelbling et al. (1998) and Sutton et al. (2018). The problem of reinforcement learning is to learn a control policy in a dynamical system, and the dynamic system can be defined as a Markov decision process (MDP, Bellman 1958).

The finite-horizon Markov decision process can be described as a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, p, r, \gamma, d_0)$, where $\mathcal{S}$ is the set of all states, $\mathcal{A}$ is the set of all actions, which can either be discrete or continuous, $p = Pr(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ is the transition dynamics, which captures the probability distribution over the next state $\mathbf{s}_{t+1} \in \mathcal{S}$ given the current state $\mathbf{s}_t \in \mathcal{S}$ and current action $\mathbf{a}_t \in \mathcal{A}$, $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function that maps the current state and action to a scalar reward signal, $\gamma \in [0, 1)$ is a discount factor, and $d_0$ define the

let $\mu_{t,j} = p(s_t = j)$

let $\xi_{t,k} = p(a_t = k)$

let $\mathcal{T}_{i,j,k} = p(s_{t+1} = i | s_t = j, a_t = k)$

$$\mu_{t+1,i} = \sum_{j,k} \mathcal{T}_{i,j,k} \mu_{t,j} \xi_{t,k}$$

initial state distribution $d_0(\mathbf{s})$ in this MDP.

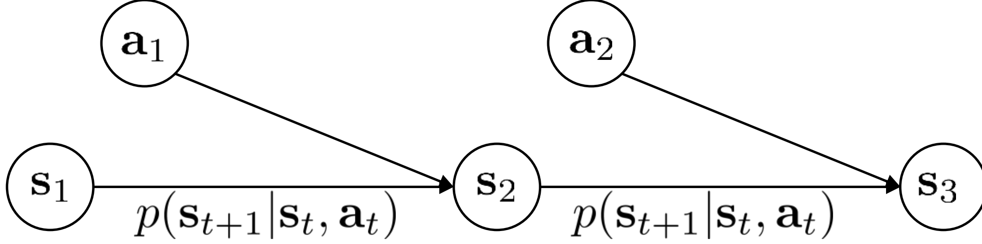


Figure 2.1: Markov decision process.

Specifically, in our problem setting, the finite-horizon MDP, the agent interacts with the environment through episodes. For every episode, the interaction starts from the time step $t = 0$ with the initial state $\mathbf{s}_0$ sampled from the initial state distribution $d_0$ and terminates at terminal state $\mathbf{s}_T$ with time step $t = T$. After an episode is terminated, the environment will reset and a new episode will start. For every time step $t$ in one episode, the agent receives the current state $\mathbf{s}_t \in \mathcal{S}$ and use its policy $\pi$, which is a probability distribution, to sample an action $\mathbf{a}_t \sim \pi(\cdot|\mathbf{s}_t)$ and execute it in the environment. In the next time step $t+1$, the environment proceeds to the next state $\mathbf{s}_{t+1} \sim p(\cdot|\mathbf{s}_t, \mathbf{a}_t)$ and emits a scalar reward signal $R_{t+1} = r(\mathbf{s}_t, \mathbf{a}_t)$. Then, $\mathbf{s}_{t+1}$ and $R_{t+1}$ will be sent to the agent. Such cycle repeats until the terminal state $\mathbf{s}_T$ is reached. We denote the value function of state under a policy $\pi$ as $v_\pi(\mathbf{s}_t)$, which can be defined recursively:

$$v_\pi(\mathbf{s}_t) = \sum_{\mathbf{a}_t} \pi(\mathbf{a}_t|\mathbf{s}_t) \sum_{\mathbf{s}_{t+1}} p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) \left[ r(\mathbf{s}_t, \mathbf{a}_t) + \gamma v_\pi(\mathbf{s}_{t+1}) \right]. \tag{2.1}$$

Similarly, we denote the action-value function of state-action pair under a policy $\pi$ as $q_\pi(\mathbf{s}_t, \mathbf{a}_t)$:

$$q_\pi(\mathbf{s}_t, \mathbf{a}_t) = \sum_{\mathbf{s}_{t+1}} p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) \left[ r(\mathbf{s}_t, \mathbf{a}_t) + \gamma v_\pi(\mathbf{s}_{t+1}) \right]. \tag{2.2}$$

In practice, we use $V_\pi$ and $Q_\pi$ to estimate $v_\pi$ and $q_\pi$, respectively.

However, as shown in Chapter 3 in our physical task setup, image observations are involved in our problem. Then, the agent does not have access

to the environment state $\mathbf{s}_t$. Instead, the agent receives a partial observation $\mathbf{o}_t$, which is a function of the environmental state. The observation space is determined by the task design, which usually attempts to incorporate enough information about the environment. For example, in vision-based tasks for agents to learn from pixels, multiple subsequent images are typically stacked together. Despite such a design, instantaneous observations may not contain enough state information about the environment. This issue can be addressed using representation learning with recurrent networks, which we do not study in this work.

## 2.2 Convolutional Neural Networks

Deep learning models are used as the function approximators in deep reinforcement learning. The two most widely used types of models are fully-connected neural networks and convolutional neural networks. When the observation is a multi-dimensional vector, fully-connected neural networks can be used, which are mostly composed of fully-connected layers. Fully-connected layers are like matrices and mathematically perform vector-matrix multiplication to the input and output a new vector. By stacking multiple fully-connected layers and non-linear activation functions applied between each layer, the model can theoretically approximate arbitrary functions. The purpose of applying non-linear activation functions is to prevent the stacked fully-connected layers from reducing to linear mapping. However, when the observation exhibits a grid pattern, such as images, convolutional neural networks need to be utilized, which imitates the visual cortex of animals to learn hierarchical and spatial features. A convolutional neural network is primarily composed of convolution layers, pooling layers, and fully-connected layers. The convolution layer is the key component of a convolutional neural network, which performs the mathematical convolution operation on grid-patterned data with a small grid of parameters called the kernel. The pooling layer performs down-sampling to reduce the spatial resolution of the input, which discards the spatial information and maintains the high-level feature patterns. As before, non-linear

activation functions are also applied after the convolution layers. A typical architecture of the convolutional neural network is shown in Figure 2.2

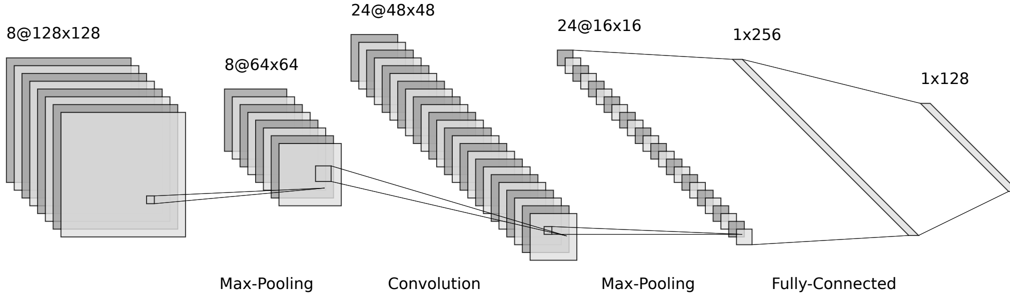


Figure 2.2: A typical convolution neural network model with both convolution layers and fully-connected layers.

However, reinforcement learning from image observation, which is our task setup shown in Chapter 3, is known to be much more challenging due to the intrinsic difficulty of learning from images as well as the instability of reinforcement learning itself. Thus, various model variants and techniques have been proposed to resolve this issue and enable reinforcement learning from images. In particular, we describe random augmentation and spatial softmax, which we use in our implementation to successfully learn from image observations. For simplicity, we use only one instance of random augmentations, yet the most effective one: random cropping (Laskin et al. 2020, Yarats et al. 2021) on image observations. Every time an image is sampled, a large patch from it will be cropped randomly and fed to the agent. This approach has been empirically demonstrated to increase performance significantly and reduce overfitting. We denote the image observation as $\mathbf{o}$ with height $H$, width $W$, and channel $C$. In this case, $\mathbf{o}$ is a tensor with shape $H \times W \times C$. We denote the cropped height and width as $\bar{H}$ and $\bar{W}$, in which $\bar{H} \leq H$ and $\bar{W} \leq W$. Then, the cropped height $h$ can be sampled from $U(0, H - \bar{H})$ and cropped width $w$ can be sampled from $U(0, W - \bar{W})$. The random cropping augmentation can be described with a Pythonic equation:

$$\text{AUG}(\mathbf{o}) = \mathbf{o}_{h:\bar{H}+h, w:\bar{W}+w,:} \tag{2.3}$$

However, it should be noted that such augmentation will only be applied to sampled mini-batch for gradient update. To sample an action to interact with the environment, $h = (H - \bar{H})/2$ and $w = (W - \bar{W})/2$ will be used to crop the central patch from the image observation. A demonstration of random cropping augmentation using DeepMind Control Suite (Tassa et al. 2020) is shown in Figure 2.3

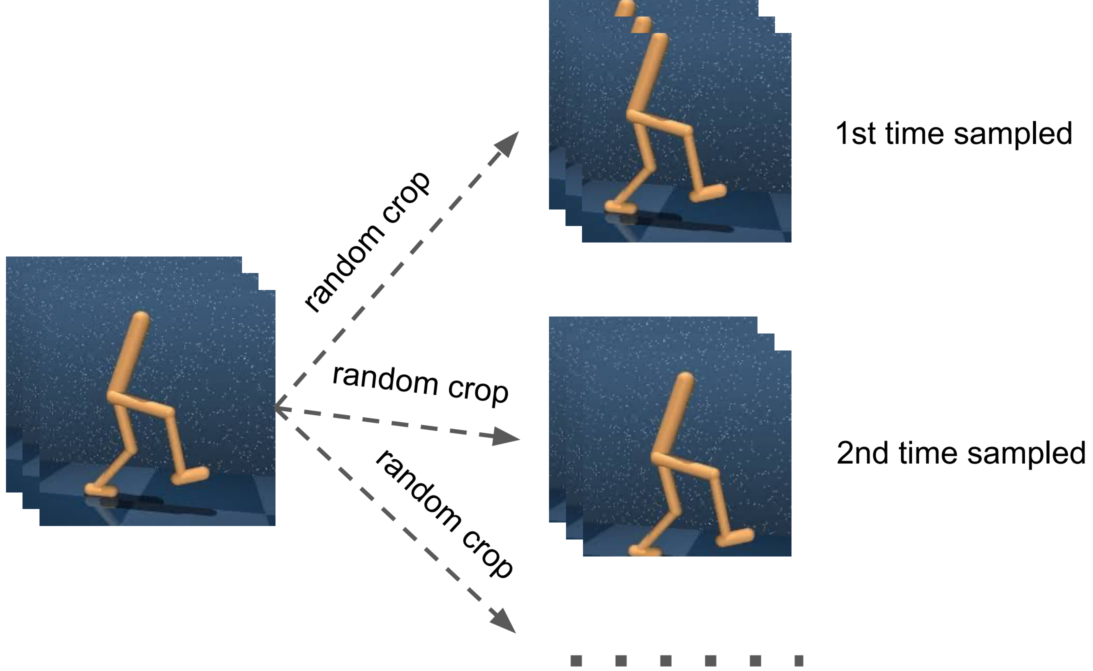


Figure 2.3: Random cropping augmentation applied to image observations.

The spatial softmax (Finn et al. 2016) exponentiates the encoding after the convolution layers and converts it into soft coordinates in each channel based on the response of activations. Such operation will discard any pattern information and only maintain spatial information, which makes training easier and it is suitable for our task setup. In spatial softmax, the latent representation is first passed to a softmax function $s_{cij} = e^{a_{cij}} / \sum_{i'j'} e^{a_{ci'j'}}$, where $a$ is the input, $s$ is the output, $c$ is the index of channel, and $(i, j)$ is the pixel coordinate. The output $s$ in each channel is a probability distribution over the location of a feature in the image. To convert this distribution to soft coor-

dinates $(f_{cx}, f_{cy})$, the expected pixel position of each feature is calculated by $f_{cx} = \sum_{ij} s_{cij} x_{ij}$ and $f_{cy} = \sum_{ij} s_{cij} y_{ij}$, where $x_{ij}$ and $y_{ij}$ is the pixel coordinate of point $(i, j)$ in the encoding. After $(f_{cx}, f_{cy})$ is obtained, other information in vector form, such as proprioceptive information, can be concatenated and fed to fully-connected layers as a whole.

## 2.3 The Soft Actor-Critic Algorithm

The Soft Actor-Critic algorithm is proposed by Haarnoja et al. (2018a), where the critic learns the action-value estimate with entropy bonus and the actor minimizes the KL-divergence between the policy distribution and exponentiated action-value estimate. Soft Actor-Critic is an off-policy, sample-efficient, and robust algorithm, which makes it particularly appealing to be utilized in real-world experiments. Later on, Lan et al. (2021) proposes the reparameterization policy gradient theorem, which provides an alternative way of understanding and deriving the policy update in Soft Actor-Critic.

Soft Actor-Critic learns a parameterized soft Q-function $Q_\theta(\mathbf{s}_t, \mathbf{a}_t)$ and a tractable policy $\pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$ with $\theta$ and $\phi$ being their neural network parameters, respectively. The output of the soft Q-function is the soft Q-value estimate, while the policy outputs the mean and variance of a Gaussian distribution of actions. If we denote the mini-batch sampled from the replay buffer $\mathcal{D}$ as $\mathcal{B}$. The soft Q-function parameters can be trained to minimize the soft Bellman residual:

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t, R_{t+1}, \mathbf{s}_{t+1}) \sim \mathcal{D}} \left[ \frac{1}{2} \left( Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - (R_{t+1} + \gamma V_{\bar{\theta}}(\mathbf{s}_{t+1})) \right)^2 \right], \tag{2.4}$$

where the soft value function is defined as:

$$V_{\bar{\theta}}(\mathbf{s}_{t+1}) = \mathbb{E}_{\mathbf{a}_{t+1} \sim \pi} \left[ Q_{\bar{\theta}}(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - \alpha \log \pi_\phi(\mathbf{a}_{t+1}|\mathbf{s}_{t+1}) \right], \tag{2.5}$$

where $\alpha$ is the temperature parameter that determines the relative importance of the entropy term. The agent maximizes the following policy objective where the action is reparameterized:

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[ \mathbb{E}_{\mathbf{a}_t \sim \pi_\phi} \left[ Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \alpha \log \pi_\phi(\mathbf{a}_t|\mathbf{s}_t) \right] \right]. \tag{2.6}$$

In an earlier version of Soft Actor-Critic (Haarnoja et al. 2018a), the temperature $\alpha$ is a constant and manually tuned across different tasks, while in a later version (Haarnoja et al. 2018b), $\alpha$ is automatically adjusted by approximating dual gradient descent:

$$J(\alpha) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[ \mathbb{E}_{\mathbf{a}_t \sim \pi_t} \left[ -\alpha \log \pi_\phi(\mathbf{a}_t|\mathbf{s}_t) - \alpha \hat{\mathcal{H}} \right] \right] \tag{2.7}$$

where the target entropy $\hat{\mathcal{H}}$ is a tunable parameter. In our implementation, we follow the automatic entropy adjustment approach.

The algorithm of a generic Soft Actor-Critic with automatic entropy adjustment is listed in Algorithm 1

**Algorithm 1** Soft Actor-Critic (generic)

| | |
|---|---|
| **initialize:** $\phi$, $\theta$, $\alpha$ | ▷ Parameters initialization |
| $\bar{\theta} \leftarrow \theta$ | ▷ Target parameters initialization |
| $\mathcal{D} \leftarrow \{\}$ | ▷ Replay buffer initialization |
| **for** each time step $t$ **do** | |
|     $\mathbf{a}_t \sim \pi_\phi(\cdot|\mathbf{s}_t)$ | ▷ Sample action from the policy |
|     $\mathbf{s}_{t+1}, R_{t+1} \sim p(\cdot|\mathbf{s}_t, \mathbf{a}_t)$ | ▷ Interact with the environment |
|     $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, R_{t+1}, \mathbf{s}_{t+1})\}$ | ▷ Add transition to replay buffer |
|     $\mathcal{B} \sim \mathcal{D}$ | ▷ Sample mini-batches |
|     $\theta \leftarrow \theta - \lambda \nabla_\theta J_Q(\theta)$ | ▷ Update Q-function parameters |
|     $\phi \leftarrow \phi - \lambda \nabla_\phi J_\pi(\phi)$ | ▷ Update policy parameters |
|     $\alpha \leftarrow \alpha - \lambda \nabla_\alpha J(\alpha)$ | ▷ Adjust temperature |
|     $\bar{\theta} \leftarrow \tau\theta + (1-\tau)\bar{\theta}$ | ▷ Update target parameters |
| **end for** | |

# Chapter 3

# Proposed Vision-based Control Environment

In this chapter, we introduce our real-world vision-based control environment, consisting of two tasks, *Reaching* and *Tracking*, which we use to investigate asynchronous reinforcement learning under the real-world setting. We first describe the physical setup of the environment and then describe how it is implemented internally to reduce the system delay and mitigate real-world challenges. Lastly, we describe the task specifications of the environment in accordance with standard reinforcement learning environments.

## 3.1 The Physical Setup

Our vision-based control environment consists of two tasks: *Reaching* and *Tracking*. The objective of the *Reaching* task is to reach arbitrary static target positions displayed on a computer monitor by a camera mounted on the wrist of the robotic arm using low-level control. In the *Tracking* task, the target is constantly moving during an episode. The physical setup of the environment is shown in Figure 3.1.

The physical setup of our environment consists of three devices, which are all connected to one workstation. First, a 6-DoF robotic arm called UR5 performs the reaching or tracking behavior. We send actuation commands to and receive proprioception from this robotic arm. The UR5 is a commercially available industrial robot manufactured by Universal Robots. This robot has

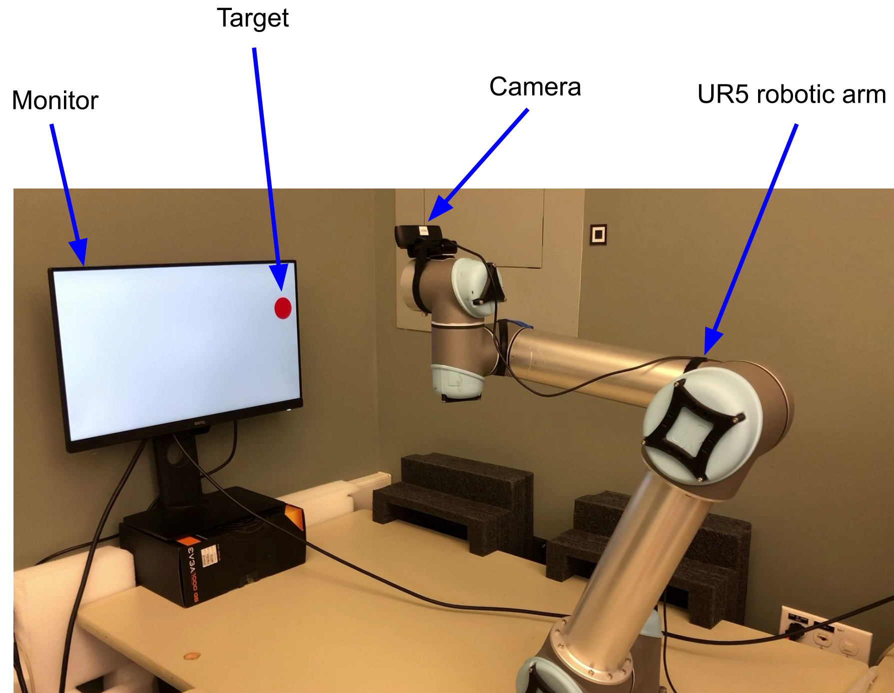


Figure 3.1: The physical setup of the environment, including the robotic arm, the wrist-mounted camera, and the monitor.

a built-in low-level programmable controller called *URControl*. The robot can be controlled with its proprietary programming language called *URScript*. After a TCP/IP connection is established, the *URScript* programs can be sent from the connected workstation to the robot controller as strings over the socket. When *URScript* programs are running on *URControl*, *URControl* streams status packets every 8ms, which is composed of sensory reading from the robot, including joint angles, joint velocities, and joint accelerations. The same 8ms is also the minimal control cycle time available on this robot. For low-level control, if no actuation command is received after 8ms, *URControl* will keep repeating the previous actuation command. Second, a computer monitor displays the target. The monitor is connected to the workstation through an HDMI cable and displays a red target on a white background. The target is either static or constantly moving in *Reaching* and *Tracking*, respectively. Third, an RGB camera captures images in front of the robotic arm to perceive the position of the target. This camera is connected to the workstation through a USB cable and is mounted on the wrist of the arm. An

image captured by the camera is shown in Figure 3.2

Figure 3.2: The $160 \times 90$ image captured by the wrist-mounted camera, where the red dot is the target.

The hardware specs of our workstation are a 16cores/32threads AMD Ryzen Threadripper 2950X, NVIDIA RTX 2080Ti with 11Gbs of RAM, and 128GBs of CPU RAM. All the results we show in the following chapters are obtained from this workstation.

The *URScript* offers two kinds of low-level control commands, position control and velocity control, though another common control approach, torque control, is not supported. We choose velocity control for UR5 as reinforcement learning with position control is not feasible. At the initial stage of reinforcement learning, the policy will generate arbitrary actions. When such actions are position control commands, the robot will behave in a violent way, leading to abrupt movements and even emergency stops.

To avoid collision with surrounding objects, we impose 3-dimensional Cartesian boundaries $0.8m \times 0.7m \times 0.7m$ to the end-effector. At every time step, *URControl* checks if safety boundaries are respected and computes velocity control actuation that returns the arm back to safety in case they are not. Additionally, to avoid the collision of the different joints of the UR5, we also impose joint angle boundaries on each joint, which works similarly to the Cartesian boundaries.

## 3.2 Environment Interactions Implementation

In the real world, time marches on regardless of the time cost of I/O and environment interaction computations, and thus the information received by the learning agent is always delayed. To minimize such systematic delays, we follow the SenseAct framework (A. R. Mahmood et al. 2018a), in which computations regarding agent-environment interactions are ordered and distributed among multiple concurrent processes to enable timely communication between the agent and environment devices with reduced latency. Based on the SenseAct framework, we initialize the three devices as three asynchronously-running processes and distribute computations between the environment interface process and the three device processes. Figure 3.3 depicts the architecture of our environment, which primarily consist of three components: 1) the environment interface process, handling agent-environment interactions and exchanging information with other device processes, 2) the devices processes, responsible for device-specific input and output, and 3) shared memory which both environment interface process and the device processes have read/write access to, to enable faster communication and avoid memory reallocation.

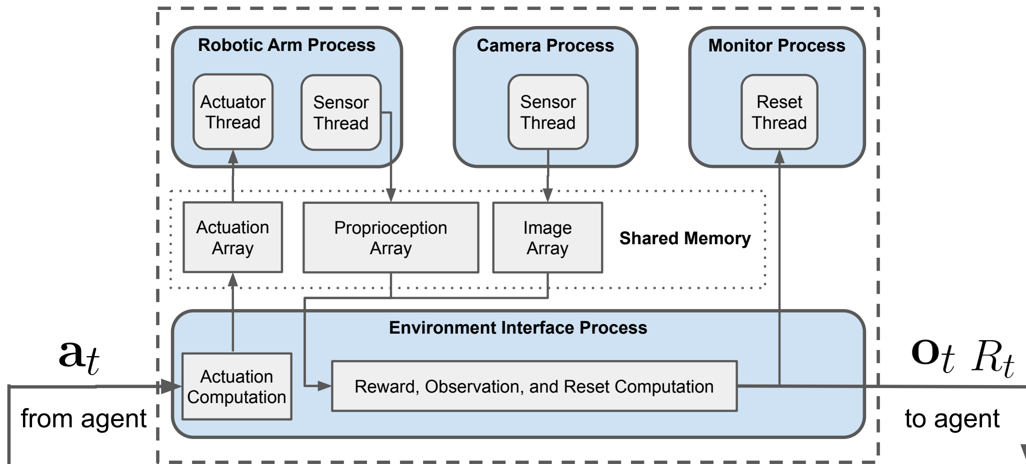


Figure 3.3: The environment architecture in this work.

The robotic arm process primarily executes the received actuation command and uses two extra I/O threads to receive actuation commands and

send proprioceptive readings. Every 8ms, the actuator thread checks from the shared actuation array and reads the newest actuation command. This actuation command will then be sent to *URControl* for execution. Meanwhile, the sensor thread will read current sensory readings from the *URControl* and write them to the shared proprioception array.

The camera process consistently captures images every 40ms and uses an I/O thread to send the captured image to the shared image array. Every 40ms, this thread writes the captured image to the shared image array. The monitor process displays the red target and uses an I/O thread that waits for the resetting command after an episode terminates to randomize the target's location, when applicable. The purpose of using asynchronous I/O threads and shared memory is to minimize the system delay caused by data I/O or data transfer between different memory allocations.

The workflow of the environment interface process can be described as the following. Every time it receives the action from the agent, it computes the corresponding actuation command based on *URScript* and writes the actuation command to the shared actuation array. Then, it reads the newest proprioception and image from the shared proprioception array and shared image array, respectively. However, as the robotic arm process and camera process are operating at different frequencies, 8ms and 40ms, the timestamps attached to the two data streams will be used to synchronize proprioceptions and images. If one of them is too old than the other, it will be discarded and wait for new data to be written to the corresponding shared data array. After the proprioceptions and images are obtained, the environment interface process will compute the reward and start to reset the environment if the terminal time step is reached. The reset function will also be executed asynchronously. At last, the environment interface process will return the subsequent observation, the reward, and the termination signal to the agent. Such cycles repeat until the preset training time step is reached. Additionally, there is an internal timer in the environment interface process to ensure the action cycle time chosen is satisfied.

## 3.3 Environment Specifications

To enable agent-environment interactions shown in Figure 3.4, it is convenient to wrap our implemented environment into a standard reinforcement learning environment so that reinforcement learning algorithms can be evaluated. Thus, we follow the environment interface used by OpenAI Gym (Brockman et al. 2016). The environment has a `step` function, which takes an action as input, and returns observation, reward, and termination signal. Additionally, the environment also has a `reset` function, which initializes a new episode.

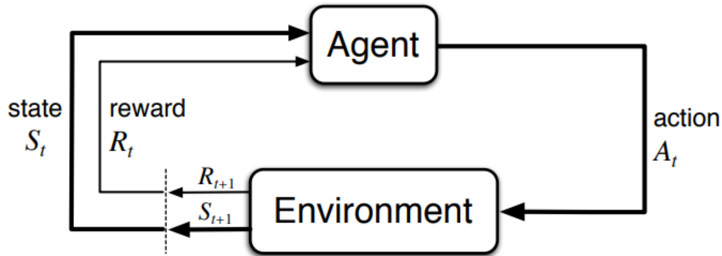


Figure 3.4: The agent-invironment interaction cycle (Sutton et al. 2018).

The observation sent to the agent contains two components, the most recent three frames of images following Mnih et al. (2013), where the image shape is either $160 \times 90 \times 3$ or $320 \times 180 \times 3$ in different experimental settings, as well as the proprioceptive information: current joint angles, current joint velocities, and previous action, where the latter is added to mitigate the negative effect of the non-Markovian property of the observation. Thus, the observation space is a tuple containing the stacked images and the proprioceptive information. The shape of the image part is either $160 \times 90 \times 9$ or $320 \times 180 \times 9$ with each element within $[0, 255]$ and the length of the proprioceptive vector is 15.

We actuate five joints excluding the wrist for the robotics arm, and the angular velocity is between $[-0.7, 0.7]$ rad/s on the five actuated joints. Thus, the action space is a vector with length 5, and each dimension is within $[-0.7, 0.7]$.

The reward function is defined as:

$$R_t = \alpha \frac{1}{hw} \sum_{i=0}^{h} \sum_{j=0}^{w} M_{ij} W_{ij} - \beta(|\pi - \sum_{n=1}^{3} \omega_n| + |\sum_{n=4}^{5} \omega_n|), \tag{3.1}$$

where $h$ and $w$ are the height and width of the image in pixels, $M$ is a $0-1$ mask matrix with shape $h \times w$ indicating whether pixels are within the color threshold of the target, $W$ is the weight matrix with shape $h \times w$, in which weights decrease quadratically from 1 to 0 from its center to edges, and $\omega_n$ is the angle of $n$th joint. The first part of the reward function encourages the agent to move closer to the target and keep the target at the center of the frame, while the second part penalizes it when twisting the joints too much. We set the coefficients $\alpha = 800$ and $\beta = 1$ for all experiments.

The difference between *Reaching* and *Tracking* is based on whether the targets move during the episode. Specifically, targets in *Reaching* are randomly generated at the beginning of each episode and stay static during the episode. Targets in *Tracking* move consistently towards a random direction and bounce from the edges. The episode length is 4 seconds, and the total number of time steps in an episode varies with the chosen action cycle time. After one episode, the environment resets by bringing the arm to a particular position in about 3 seconds.

# Chapter 4

# Asynchronous Learning Architecture

In this chapter, we first introduce our implementation of the Soft Actor-Critic algorithm and the neural network model architecture. Then we introduce the sequential learning architecture utilized in most open-source implementations and the existing asynchronous learning architecture in some works that adopt the real-world setting. Based on the two existing architectures, we describe our extended asynchronous learning architecture and discuss how we can achieve the former two architectures based on our unified implementation. In our experiments, we use our implementation to emulate the three different architectures and evaluate their performance. Lastly, we analyze the benefits of our proposed learning architecture when learning from images in terms of learning cycle time and action cycle time.

## 4.1 The Soft Actor-Critic Implementation

Our Soft Actor-Critic implementation follows Yarats et al. (2019), with two major architectural improvements. First, we use spatial softmax (Finn et al. 2016) to convert the encoding space into soft coordinates to track the target more precisely and remove redundant information. Second, we apply random cropping (Yarats et al. 2021, Laskin et al. 2020) to augment images in mini-batches to learn more robust representations given our limited amount of observations.

The neural network architecture we used is shown in Figure 4.1. The stacked image observations are first processed by the CNN encoder and then flattened by the spatial softmax layer. The flattened coordinate vector is concatenated with the proprioceptive vector and fed into the actor MLP and critic MLP, respectively. According to Yarats et al. (2019), the non-stationary gradients from the actor loss can impede encoder learning. Thus, we prevent the gradients from actor to update the CNN encoder and use only the critic loss to update the CNN encoder.

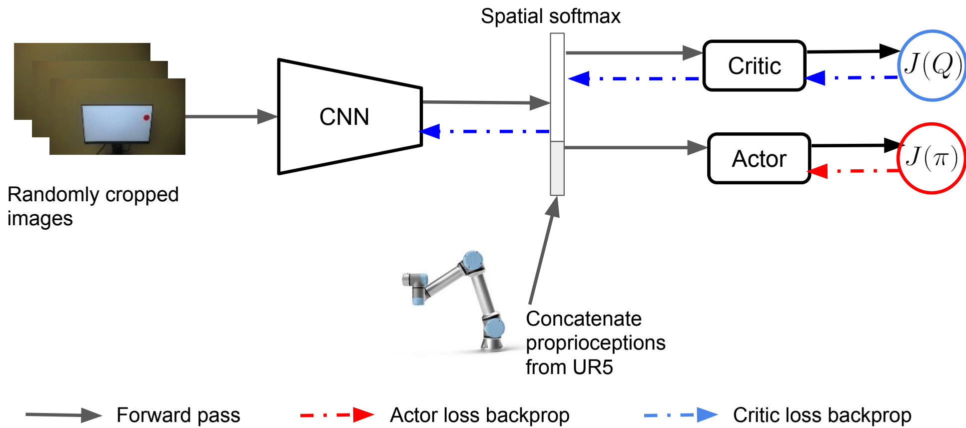


Figure 4.1: The neural network architecture used in the experiments. The solid arrows show the directions of the forward pass, and the dashed arrows show the directions of the backward pass.

For the actor-critic network, we employ a similar critic architecture as used in TD3 (Fujimoto et al. 2018), which uses double Q-networks to mitigate the issue of over-estimation in Q-value estimation. Each critic is parametrized as a 3-layer MLP with ReLU activations after each layer except for the last, which outputs the Q-value. The actor is also a 3-layer MLP with ReLU as the activation function. The actor network outputs the mean and diagonal covariance of a Gaussian distribution, and the action is sampled from this distribution. The hidden dimension is set to 1024 for both the critic and the actor.

For the convolution network, we employ kernels of size $3 \times 3$ with 32 channels for all the four convolution layers in our encoder network. The stride is set to 2 except for the last layer, whose stride is set to 1. The output of

the convolution layers is fed into a spatial softmax layer (Finn et al. 2016) and then converted into soft coordinates as a vector of length 64. The encoder network is shared between the actor and critic networks. However, only the gradients from the critic are allowed to update the shared convolution layers. The gradients from the actor are truncated before back-propagating to the shared convolution layers. As the critical point of this model is to maintain spatial information, max-pooling layers are not used in our model.

## 4.2 The Sequential Learning Architecture

Most available open-source implementations of RL algorithms process all computations sequentially, usually in a single process. Such implementations are simple and effective in simulated environments. For Soft Actor-Critic, or more generally, RL algorithms with replay buffers and per-step updates, their computational order is shown in Figure 4.2

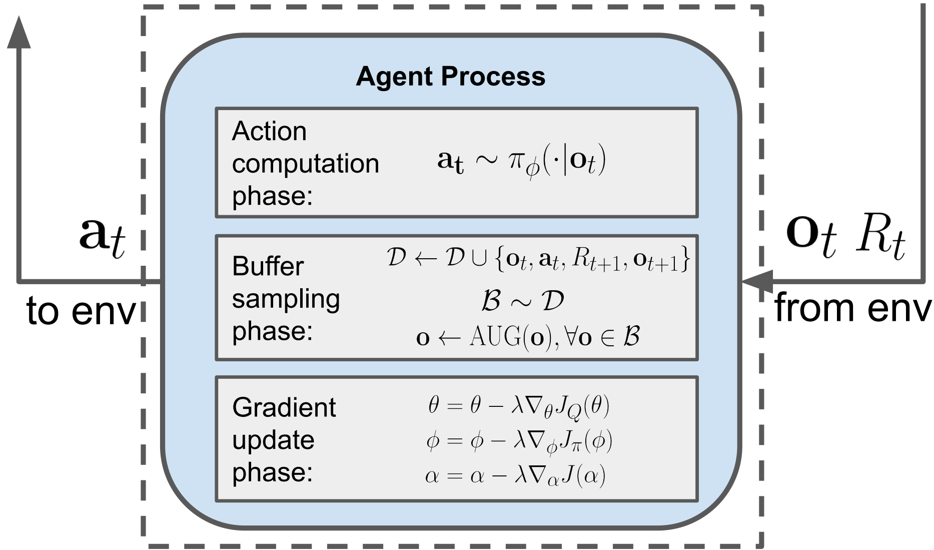


Figure 4.2: Overview of the sequential learning architecture.

After receiving the next observation from the environment, the agent needs to finish all the three phases of computations: replay buffer adding and sampling, gradient update, and action computation, before it can send the action

selected to the environment. We show our implementation sequential Soft Actor-Critic in Algorithm 2.

**Algorithm 2** Soft Actor-Critic (the sequential implementation)

```
initialize: φ, θ, α                                  ▷ Parameters initialization
θ̄ ← θ                                                ▷ Target parameters initialization
D ← {}                                               ▷ Replay buffer initialization
for each time step t do
    a_t ∼ π_φ(·|o_t)                                 ▷ Sample action from the policy
    o_{t+1}, R_{t+1} ∼ p(·|o_t, a_t)                 ▷ Interact with the environment
    D ← D ∪ {(o_t, a_t, R_{t+1}, o_{t+1})}           ▷ Add transition to replay buffer
    B ∼ D                                            ▷ Sample mini-batches
    o ← AUG(o), ∀o∈ B                                ▷ Randomly crop images
    θ ← θ − λ∇_θ J_Q(θ)                              ▷ Update Q-function parameters
    φ ← φ − λ∇_φ J_π(φ)                              ▷ Update policy parameters
    α ← α − λ∇_α J(α)                                ▷ Adjust temperature
    θ̄ ← τθ + (1 − τ)θ̄                                ▷ Update target parameters
end for
```

This computation arrangement is valid in simulated environments because the environment will pause internally during the computation. So the subsequent observation after 1ms or 1 second of computation time will remain essentially the same. However, in real-world environments, time marches on during these computations. For a robotic system, the total time spent during learning updates, together with the time needed for agent-environment interaction, becomes the effective action cycle time. Unfortunately, such action cycle time is usually much longer than the desirable action cycle time and potentially reduces the agent's observations and responsiveness. One practical way of applying such implementations in the real world is to manually find the minimal action cycle time adequate for the computations, as we show in Table 5.2.

## 4.3 The Existing Asynchronous Learning Architecture

To decouple learning updates from agent-environment interactions, asynchronous reinforcement learning has been proposed in (Gu et al. 2017, Yahya et al. 2017,

Haarnoja et al. 2018c). In Figure 4.3, we show the asynchronous learning architecture proposed by Haarnoja et al. (2018c).

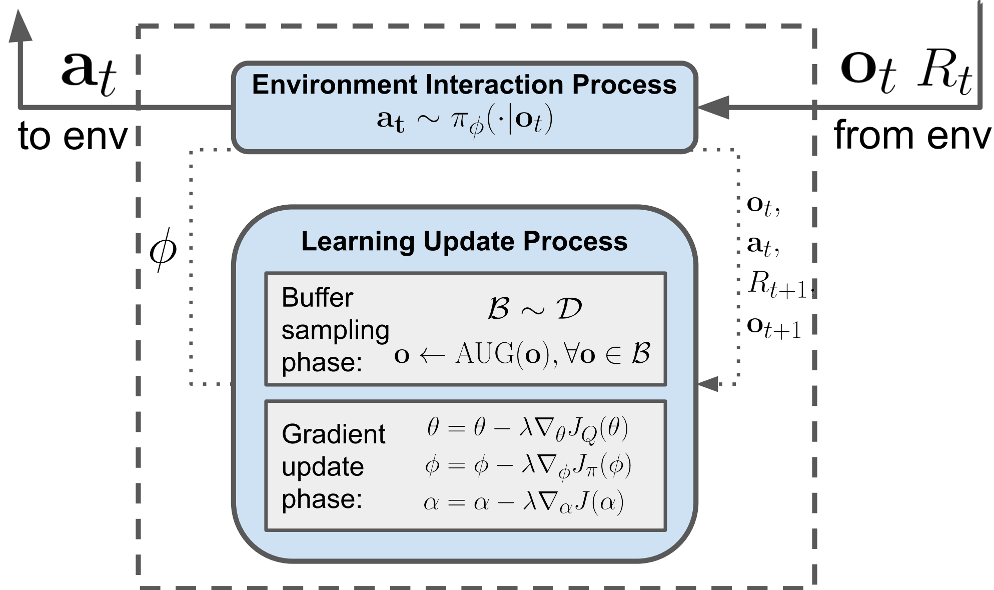


Figure 4.3: Overview of the existing asynchronous learning architecture.

Such architectures are primarily composed of two asynchronously running processes or two computers that can communicate with each other. The environment interaction process is responsible for agent-environment interactions and consistently sends collected transition data to the learning update process. Meanwhile, the learning update process runs in the background, storing the received transition data, sampling mini-batches from the replay buffer, performing the gradient update, and synchronizing updated parameters with the environment interaction process. The algorithm for the two processes is shown in Algorithms 3.

The benefit of such architectures is that no learning is involved in the environment interaction process, and the minimal action cycle time only depends on the hardware capabilities and the time needed for policy inference and it does not scale with the computation time. In this case, it is much easier to choose an action cycle time that is desirable for reinforcement learning.

**Algorithm 3** Soft Actor-Critic (the existing asynchronous learning implementation)

**Process 1: Environment Interaction**
**for** each time step $t$ **do**
$\mathbf{a}_t \sim \pi_\phi(\cdot|\mathbf{o}_t)$ ▷ Sample action from the policy
$\mathbf{o}_{t+1}, R_{t+1} \sim p(\cdot|\mathbf{o}_t, \mathbf{a}_t)$ ▷ Interact with the environment
Send $\{(\mathbf{o}_t, \mathbf{a}_t, R_{t+1}, \mathbf{o}_{t+1})\}$ to $\mathcal{D}$ ▷ Send transition to replay buffer
**end for**

**Process 2: Learning Update**
**initialize** $\phi$, $\theta$, $\alpha$ ▷ Parameters initialization
$\bar{\theta} \leftarrow \theta$ ▷ Target parameters initialization
$\mathcal{D} \leftarrow \{\}$ ▷ Replay buffer initialization
**while** training time remains **do**
$\mathcal{B} \sim \mathcal{D}$ ▷ Sample mini-batches
$\mathbf{o} \leftarrow \text{AUG}(\mathbf{o}), \forall \text{o} \in \mathcal{B}$ ▷ Randomly crop images
$\theta \leftarrow \theta - \lambda \nabla_\theta J_Q(\theta)$ ▷ Update Q-function parameters
$\phi \leftarrow \phi - \lambda \nabla_\phi J_\pi(\phi)$ ▷ Update policy parameters
$\alpha \leftarrow \alpha - \lambda \nabla_\alpha J(\alpha)$ ▷ Adjust temperature
$\bar{\theta} \leftarrow \tau\theta + (1-\tau)\bar{\theta}$ ▷ Update target parameters
**end while**

## 4.4 Our Proposed Asynchronous Learning Architecture

The existing asynchronous architecture runs learning updates and environment interaction asynchronously, but inside the learning update, the replay buffer sampling and gradient updates are still processed sequentially. Though this might be sufficient when the observations are low-dimensional vectors, it may not be efficient enough in our tasks, especially in the high-resolution and large mini-batch setting. We extend the existing architecture by separating the replay buffer sampling and gradient updates into two processes and run them in a semi-asynchronous way. We call it semi-asynchronous as the replay buffer sampling must be prior to the gradient update, so the two processes could not be run entirely asynchronously. However, they still run in parallel. Such semi-asynchronous execution shortens the cycle time for learning updates from the sum of buffer sampling and gradient updates times to the maximal of the two. The increase in learning-update frequency is substantial and can be as large

as twice when they both take an extended and similar amount of time.

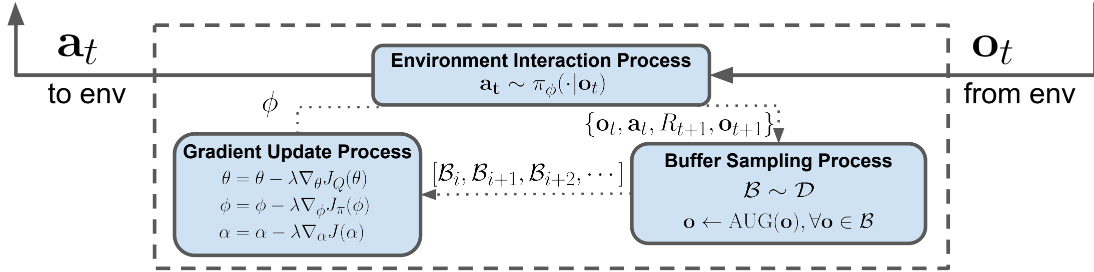


Figure 4.4: Overview of our extended asynchronous learning architecture.

Our architecture consists of three processes that run in parallel: 1) the environment interaction process, which interacts with the environment interface process, collects transition data, and stores it in the replay buffer, 2) the buffer sampling process, which stores transition data, samples a mini-batch, converts the mini-batch to CUDA tensors, and apply random augmentation to them, and 3) gradient update process, which performs gradient updates and shares policy parameters with environment interaction process. The algorithm for the three processes is shown in Algorithms 4.

## 4.5 Computational Comparison of the Three Architectures

We incorporate the three architectures in a unified implementation with three different components. Based on this implementation, we can achieve three different degrees of asynchronous execution. If we run all three components sequentially, it will correspond to most open-source SAC implementations as shown in Figure 4.2, and we call it *Seq-SAC*. If we only run environment interactions asynchronously but run replay buffer sampling and gradient update sequentially, we will get an architecture proposed previously in other work, as shown in Figure 4.3, and we call it *Async-SAC-1*. If we run environment interactions asynchronously and run replay buffer sampling and gradient update in parallel, it will correspond to our extended asynchronous learning architecture as shown in Figure 4.4, and we call it *Async-SAC-2*.

**Algorithm 4** Soft Actor-Critic (our proposed asynchronous learning implementation)

**Process 1: Environment Interaction**

**for** each time step $t$ **do**

$\mathbf{a}_t \sim \pi_\phi(\cdot|\mathbf{o}_t)$ ▷ Sample action from the policy

$\mathbf{o}_{t+1}, R_{t+1} \sim p(\cdot|\mathbf{o}_t, \mathbf{a}_t)$ ▷ Interact with the environment

send $\{(\mathbf{o}_t, \mathbf{a}_t, R_{t+1}, \mathbf{o}_{t+1})\}$ to $\mathcal{D}$ ▷ Send transition to replay buffer

**end for**

**Process 2: Buffer Sampling**

$\mathcal{D} \leftarrow \{\}$ ▷ Replay buffer initialization

**while** training time remains **do**

$\mathcal{B} \sim \mathcal{D}$ ▷ Sample mini-batches

$\mathbf{o} \leftarrow \text{AUG}(\mathbf{o}), \forall \text{o} \in \mathcal{B}$ ▷ Randomly crop images

send $\mathcal{B}$ to $\mathcal{I}$ ▷ Send mini-batches to the shared queue

**end while**

**Process 3: Gradient Update**

**initialize** $\phi$, $\theta$, $\alpha$ ▷ Parameters initialization

$\bar{\theta} \leftarrow \theta$ ▷ Target parameters initialization

**for** every B received **do**

$\theta \leftarrow \theta - \lambda \nabla_\theta J_Q(\theta)$ ▷ Update Q-function parameters

$\phi \leftarrow \phi - \lambda \nabla_\phi J_\pi(\phi)$ ▷ Update policy parameters

$\alpha \leftarrow \alpha - \lambda \nabla_\alpha J(\alpha)$ ▷ Adjust temperature

$\bar{\theta} \leftarrow \tau\theta + (1-\tau)\bar{\theta}$ ▷ Update target parameters

**end for**

The computation flow of the three architectures and the benefit of asynchronous learning in terms of shorter cycle times are shown in Figure 4.5.

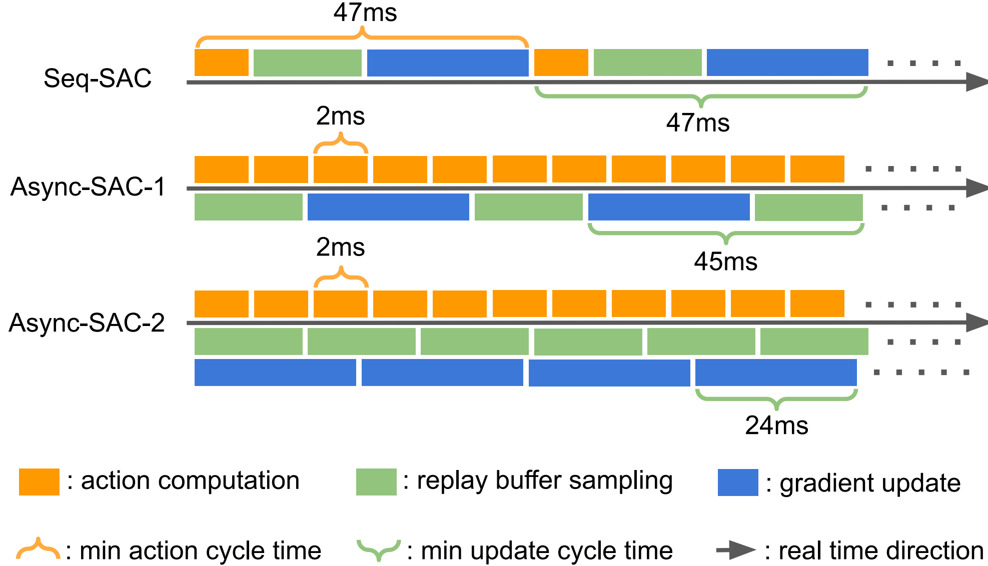


Figure 4.5: The computational flow over time of the three versions of SAC. For plotting purposes, the relative length of each block may not reflect the relative computation time.

Specifically, we show the computational time spent by action computation, replay buffer sampling, and gradient update. The amount of times for these computations given around the colored boxes are representative of real computation times, as they are actual measurements in one of our experimental settings. The measurements for all settings are given in Table 5.2. From *Seq-SAC* to *Async-SAC-1*, the minimal action cycle time decreases substantially, but the update cycle time almost remains the same. However, from *Async-SAC-1* to *Async-SAC-2*, even the update cycle time decreases significantly. Therefore, our architecture can perform learning updates faster than the existing architecture, which can be crucial when updates are computationally expensive.

# Chapter 5

# Experimental Setup

This chapter introduces the three different experimental settings: baseline, high-resolution, and large mini-batch, which we use to compare sequential and asynchronous reinforcement learning. We also illustrate how the action cycle time is chosen for *Seq-SAC*, which requires different action cycle times in different settings due to the sequentially arranged computation. Then, we introduce some modifications to standard experiment methodology due to the constraints of experimenting on a physical robot.

## 5.1 Three Experimental Settings

In our experiments, we compare sequential and asynchronous learning under different settings with different update costs. The first setting uses relatively cheap computations, where the mini-batch size is 128, and the image size is $160 \times 90 \times 3$. In this setting, we would like to explore when the action cycle time between sequential and asynchronous reinforcement learning are similar, whether the agent will benefit from asynchronous learning updates. The next two settings use increased computational cost. We attribute the increased computational cost to two factors, higher-dimensional sensory input or large mini-batch, which are both common scenarios for real-world applications. In some situations, such as surgical robots, high-resolution image observation is a must, while in other situations, high-resolution image observations are not necessary, but a reinforcement learning agent can always benefit from a large mini-batch due to the intrinsic high-variance gradients in deep reinforcement

|  | Baseline | High-resolution | Large mini-batch |
|---|---|---|---|
| Image size | $160 \times 90$ | $320 \times 180$ | $160 \times 90$ |
| Mini-batch size | 128 | 128 | 512 |

Table 5.1: Summary of three experimental settings.

learning. Thus, in the second setting, the mini-batch size is 128, and the image size is $320 \times 180 \times 3$. In the third setting, the mini-batch size is 512, and the image size is $160 \times 90 \times 3$. For these two settings, we would like to investigate how the performance of *Seq-SAC* degrade in response to the increased computation time, as well as how the two variants of asynchronous learning architectures would perform differently. We call these three settings: *baseline*, *high-resolution*, and *large mini-batch*, respectively. It should be noted that, due to random cropping augmentation, the actual image sizes fed into the model are $156 \times 88 \times 3$ and $314 \times 176 \times 3$. In Table 5.1, we show our three experimental settings.

## 5.2 Choosing Action-Cycle Time

In all settings, both asynchronous learning architectures use 40ms action cycle time as it is the default choice for reaching with UR5 in A. R. Mahmood et al. (2018b). However, *Seq-SAC* needs different action cycle times in different settings so that learning updates can fit into them. To choose the minimal affordable action cycle time for *Seq-SAC*, we record the time cost of each component in our implementation, as shown in Table 5.2, and choose the action cycle time accordingly. It should be noted that the exact time measured is implementation and hardware-specific, but their relative length should be applicable to algorithms with replay buffer and per-step updates.

As the action cycle time needs to be aligned with the hardware cycle time of 8ms of the robotic arm and the cycle time of 40ms of the camera, we use 80ms for the baseline setting. Though it is possible to use action cycle time 40ms for the *seq-SAC* in this setting, it will require a tiny mini-batch size, which is infeasible for SAC training. For the high-resolution setting, we choose 120ms, and for the large mini-batch setting, we choose 200ms. The episode length

| | Baseline | High-resolution | Large mini-batch |
|---|---|---|---|
| Action computation | 2ms | 3ms | 2ms |
| Replay buffer sampling | 21ms | 44ms | 86ms |
| Gradient update | 24ms | 65ms | 68ms |
| Environment computation | 5ms | 6ms | 5ms |
| Total | 52ms | 118ms | 161ms |

Table 5.2: The computation time of each component in *Seq-SAC* measured in our three experimental settings.

of *Reaching* and *Tracking* are both 4 seconds. In this case, the action cycle time of 40ms and 80ms will result in 100 and 50 time steps in one episode, respectively.

## 5.3 Experiment Methodology

Because of the variable action cycle time and the constraints of a physical robot, we make the following modifications to the standard experiment methodology. First, using different action cycle times under fixed training time steps can lead to substantially different actual training times. To avoid this issue, we use 2-hour wall time as the total training time, regardless of the actual number of time steps the agent takes. Second, the effective number of time steps in one episode could be shortened due to the prolonged action cycle time. Thus, we scale the reward proportional to the action cycle time to make episode returns comparable. Third, running a separate evaluation phase on a physical robot to measure the performance of the agent without exploration can be time-consuming. Instead, we report the agent's online performance as an evaluation. Fourth, at the beginning of each run, there are 1000 time steps during which the agent executes a random policy to initialize replay buffer. These additional 1000 time steps are neither counted in the 2-hour training time nor shown in the learning curve.

Hyper-parameter search on a physical robot can lead to safety issues and damage the robot. In our experiment, we follow the hyper-parameters used in Yarats et al. (2019), whose effectiveness has been empirically demonstrated in simulated environments. We use the same hyper-parameters across different

versions of SAC and different experimental settings. The complete list of hyper-parameters used is given in Table 5.3. Actor update frequency and critic target update frequency mean how frequent the two update with respect to the critic update, as the critic updates at every gradient update.

| Parameters | Value |
|---|---|
| optimizer | Adam |
| actor learning rate | 3e-4 |
| critic learning rate | 3e-4 |
| encoder learning rate | 1e-3 |
| temperature learning rate | 1e-4 |
| actor update frequency | 2 |
| critic target update frequency | 2 |
| critic target update ratio | 0.01 |
| encoder target update ratio | 0.05 |
| initial temperature | 0.1 |
| initial time steps | 1000 |
| discount factor | 0.99 |
| replay buffer size | 1e5 |
| mini-batch size | {128, 512} |

Table 5.3: SAC Hyperparameters

# Chapter 6

# Experimental Results with The Sequential Learning Architecture

In this chapter, we show the experimental results of *Seq-SAC* in our three experiment settings and give analysis to the results obtained.

## 6.1 Training Results

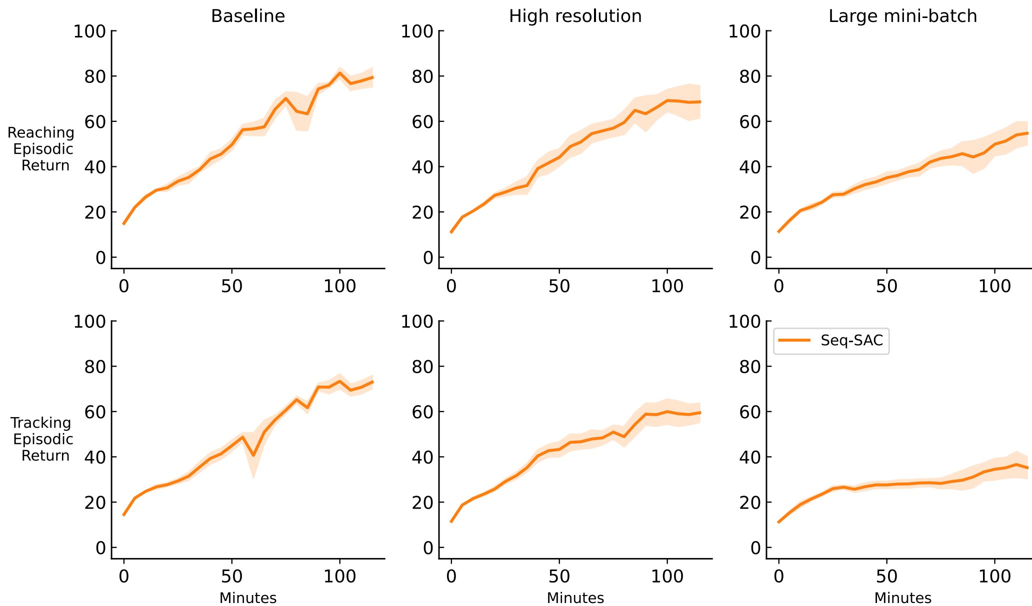


Figure 6.1: The learning curves of *Seq-SAC*

In Figure 6.1, we show the learning curves based on undiscounted episodic

returns. Each learning curve is obtained with five independent runs, and each run takes two hours. The shaded areas in the learning curves represent the standard error. The same settings are applied in Chapter 7 and 8.

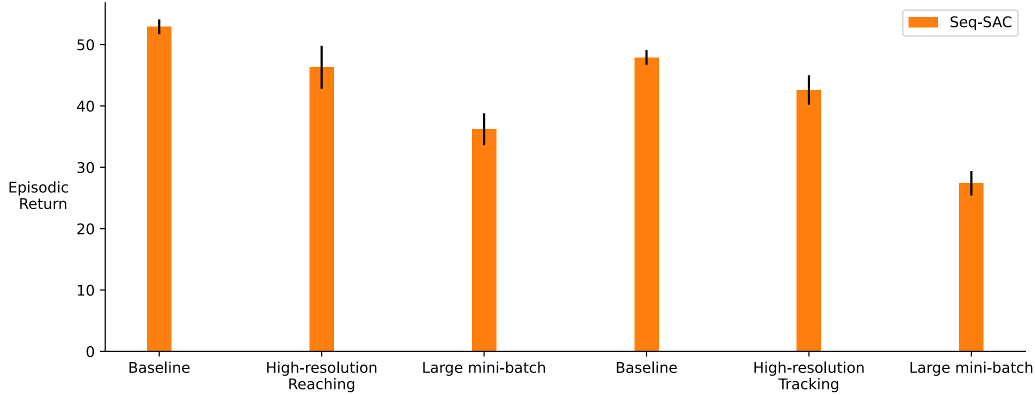


Figure 6.2: The overall performance of *Seq-SAC*

In Figure 6.2, we show the overall performance of *Seq-SAC*, which is calculated by averaging returns over the whole learning period. The error bar shows the standard error over five independent runs.

## 6.2 Evaluation Results

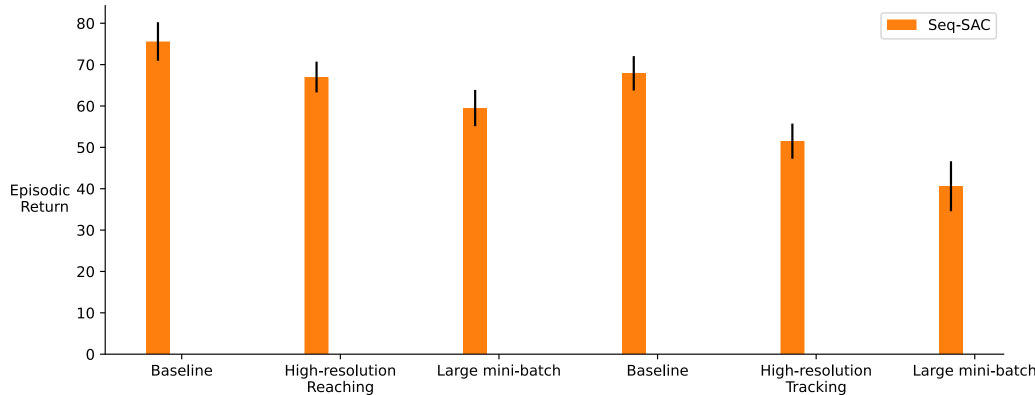


Figure 6.3: The evaluation performance of *Seq-SAC*

In Figure 6.3, we show the evaluation performance of *Seq-SAC*. The evaluation performance is obtained with the learned model after two hours of training with exploration noise turned off. Since we have five independent runs in each

setting, the result shown is the average performance of the models obtained in the five independent runs. For each model, its performance is evaluated with the averaged return over 10 episodes. The same settings are applied in Chapter 7 and 8.

## 6.3 Results Analysis

Generally, *Seq-SAC* achieved satisfactory performance in both *Reaching* and *Tracking* in the baseline setting. This indicates that when the computation cost of learning updates is relatively low and the effective action cycle time is not significantly longer than the desirable action cycle time, implementations with sequentially arranged computations can perform well. However, from the baseline setting to the large mini-batch setting, the overall trend is that the performance degraded with the increase of the computation time. This is due to the longer action cycle time, 120ms in the high-resolution setting and 200ms in the large mini-batch setting. The excessively long action cycle time substantially reduces the number of observations and gradient updates, which causes performance degradation. Another interesting observation here is that, from the baseline setting to the large mini-batch setting, the performance declined more in *Tracking* than in *Reaching*, which is likely due to the reduced responsiveness because of the excessively long action cycle time. In conclusion, when the computation cost of learning updates is low, implementations with sequentially arranged computations can perform well. However, increasing the computation cost will increase the action cycle time proportionally, which reduces the number of observations, gradient updates, and the responsiveness of the agent and eventually degrades the performance.

# Chapter 7

# Experimental Results with The Existing Asynchronous Learning Architecture

In this chapter, we present the experimental results of *Async-SAC-1* and its analysis. Results of *Seq-SAC* are also shown for comparison.

## 7.1 Training Results

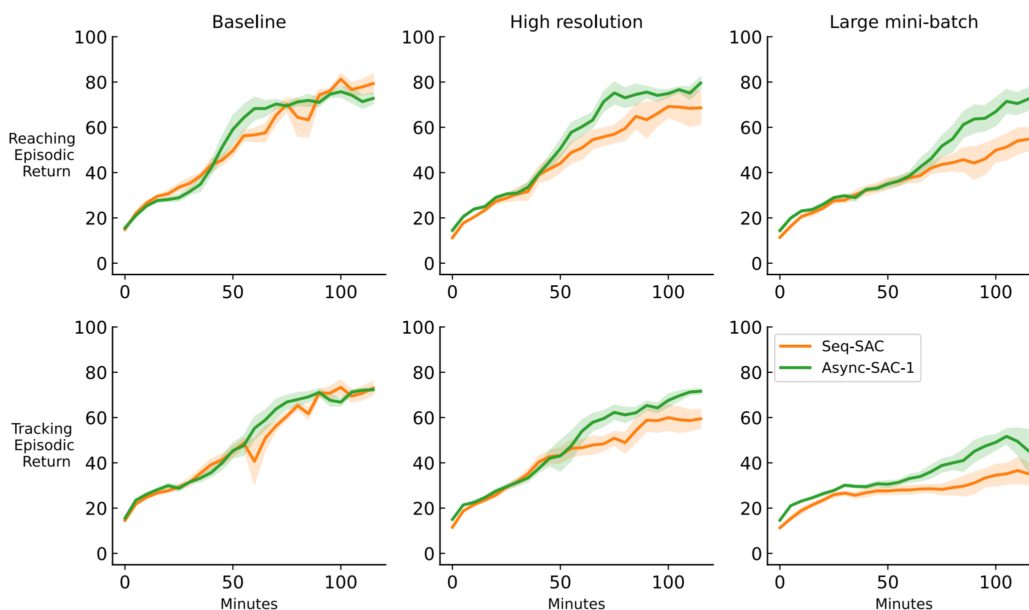


Figure 7.1: The learning curves of *Seq-SAC* and *Async-SAC-1*.

Figure 7.1 shows the learning curves of *Async-SAC-1* and Figure 7.2 shows

the average returns over the whole learning period. It should be noted that *Async-SAC-1* could make gradient updates too frequently in the baseline setting, which leads to early convergence to a sub-optimal policy. Thus, we constraint the gradient update frequency in *Async-SAC-1* to be one per one environment interaction.

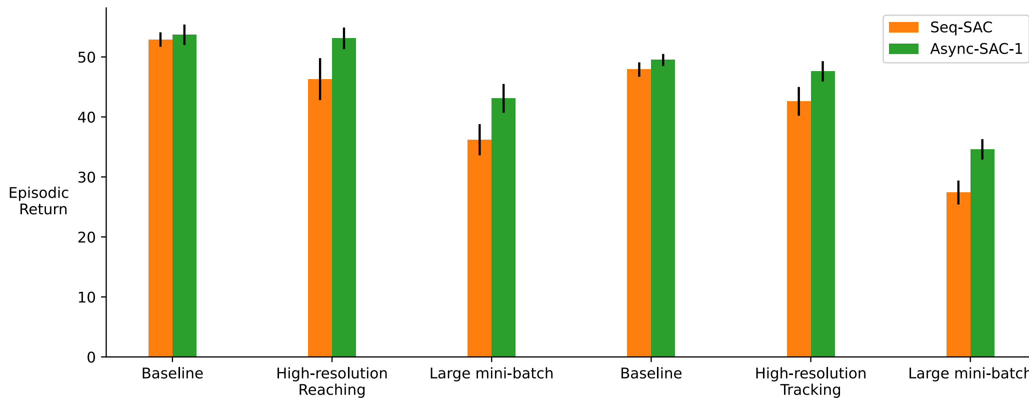


Figure 7.2: The overall performance of *Seq-SAC* and *Async-SAC-1*

## 7.2 Evaluation Results

In Figure 7.3, we show the evaluation performance of *Seq-SAC* and *Async-SAC-1*. In the baseline setting, *Seq-SAC* and *Async-SAC-1* perform similarly. However, in the other two settings, *Async-SAC-1* substantially outperforms *Seq-SAC*.

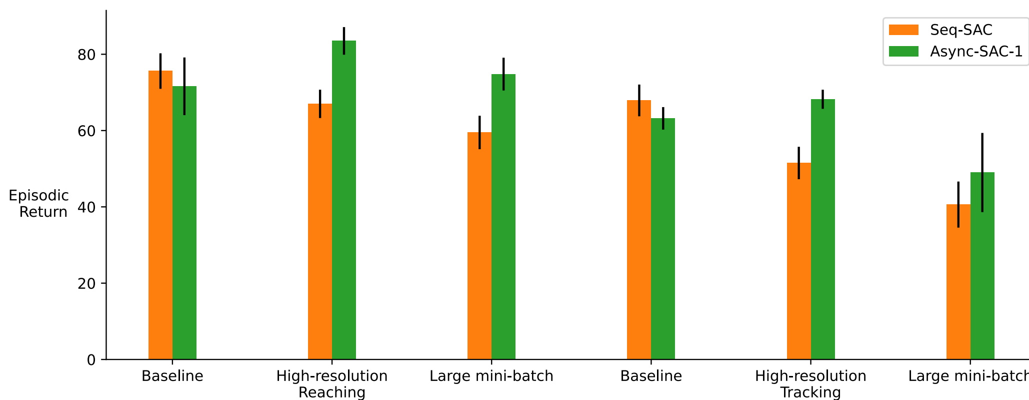


Figure 7.3: The evaluation performance of *Seq-SAC* and *Async-SAC-1*

## 7.3 Results Analysis

In the baseline setting, *Async-SAC-1* performed similarly to *Seq-SAC*. *Async-SAC-1* uses 40ms as its action cycle time while *Seq-SAC* uses 80ms and the longer action cycle time reduces the number of observations and gradient updates. However, even with the reduced observations and gradient updates, the similar performance implies the importance of action repeat, which has been used extensively in DeepMind Control Suite. In our setting, 80ms action cycle time corresponds to 40ms action cycle time with action repeat set to 2. However, the performance starts to diverge in the high-resolution setting, as *Async-SAC-1* essentially maintained the performance in the baseline setting, while the performance of *Seq-SAC* declined substantially. This demonstrates the benefits of asynchronous reinforcement learning, as it decouples environment interactions and learning updates so that the action cycle time can be chosen independently from the time cost of learning updates. Unfortunately, in the large mini-batch setting, there is a substantial degradation in performance, which indicates some architectural deficiencies with *Async-SAC-1* when the computational cost becomes extremely high. In conclusion, the existing asynchronous reinforcement architectures can maintain the performance and action cycle time regardless of the time cost of learning updates, but certain architectural improvement is needed to improve its learning update efficiency in computationally extensive settings.

# Chapter 8

# Experimental Results with Our Proposed Asynchronous Learning Architecture

In this chapter, we present the experimental results of *Async-SAC-2* and its analysis. Results of *Seq-SAC* and *Async-SAC-1* are also shown for comparison.

## 8.1 Training Results

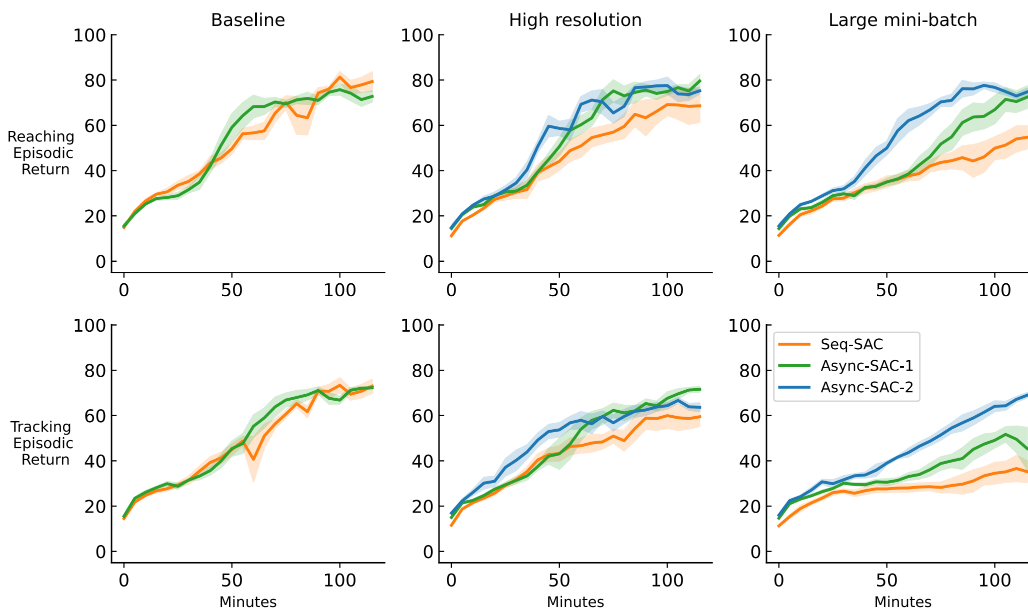


Figure 8.1: The learning curves of *Async-SAC-2*, *Async-SAC-1*, and *Seq-SAC*

Figure 8.1 shows the learning curves of *Async-SAC-1* and Figure **??** shows

the average returns over the whole learning period. Similar to *Async-SAC-1*, *Async-SAC-2* would also perform gradient updates too frequently in the baseline setting, and we also constraint its update frequency to be one per one environment interaction. However, in this case, *Async-SAC-2* and *Async-SAC-1* reduce to the same. So we use the results from *Async-SAC-1* in the baseline setting for *Async-SAC-2* without rerunning the experiments.

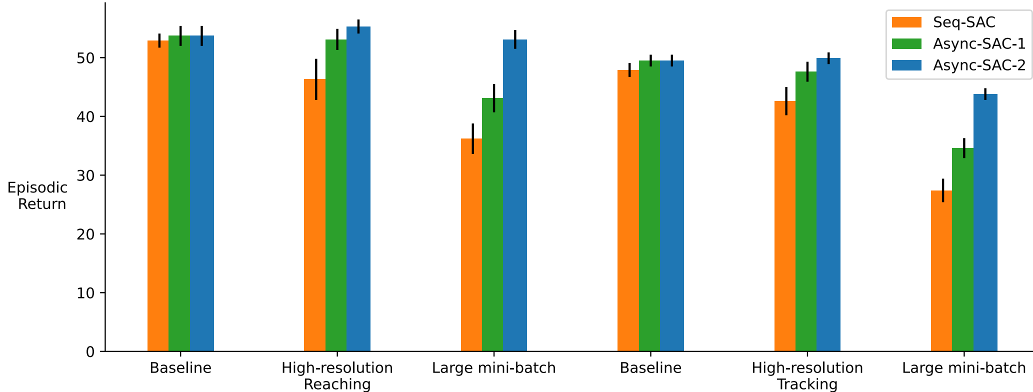


Figure 8.2: The overall performance of *Async-SAC-2*, *Async-SAC-1*, and *Seq-SAC*

## 8.2 Evaluation Results

In Figure 8.3, we show show the evaluation performance of *Seq-SAC*, *Async-SAC-1*, and *Async-SAC-2*. *Async-SAC-2* largely maintained the performance and has the smallest overall standard error different across experimental settings.

## 8.3 Results Analysis

*Async-SAC-2* performed consistently well in different settings and tasks and performed significantly better than the other two in the large-mini-batch setting, indicating that our extended architecture is more robust across different computation costs. In general, both variants of asynchronous reinforcement learning architectures performed substantially better than their sequential counterparts, indicating the real-world benefits of asynchronous reinforcement

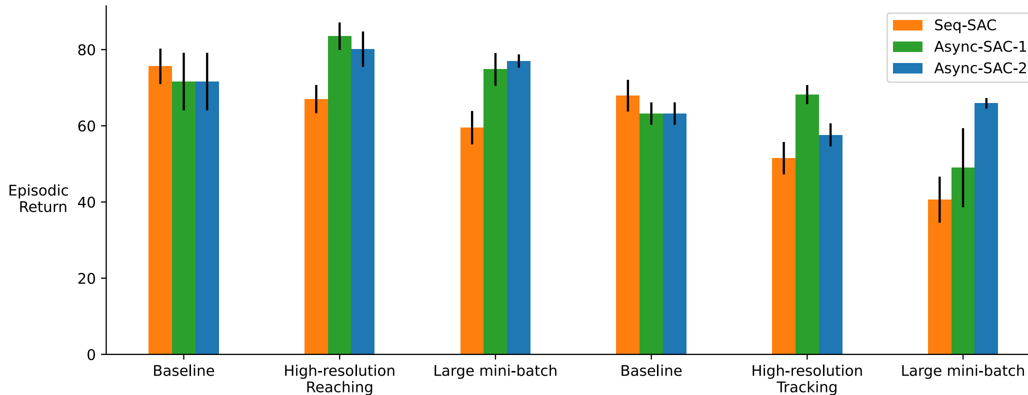


Figure 8.3: The evaluation performance of *Async-SAC-2*, *Async-SAC-1*, and *Seq-SAC*

learning. Despite the advantages of asynchronous, one drawback of asynchronous learning is that it could make learning updates faster than what is needed and lead to problems like early convergence. A certain degree of manual tunning on the update frequency is still needed.

Our empirical results show that sequential and asynchronous learning can perform similarly when learning updates are relatively cheap. However, when the learning updates become expensive due to higher resolution of images or larger mini-batch sizes, the performance of sequential learning diminishes as its action cycle time is prolonged excessively, which significantly reduces the number of observations and the agent's responsiveness. Consequentially, our asynchronous learning system performs substantially better than sequential learning in this case. We also show that when the mini-batches are considerably large, the additional parallelization of replay buffer sampling and gradient updates provides a significant performance improvement. Besides the better performance, two clear advantages of asynchronous learning are illustrated through our experiments: 1) the minimal action cycle time achievable by asynchronous learning can be much shorter than that of sequential learning, which provides more flexibility when learning in the real world, and 2) unlike sequential learning, a constant action cycle time can always be maintained by asynchronous learning regardless of the time cost of learning updates.

## 8.4 Learned Behavior

Figure 8.4 shows the learned reaching and tracking behaviors in one episode of *Async-SAC-2* after two hours of training. As can be seen, *Async-SAC-2* learned smooth and precise behaviors in both tasks.

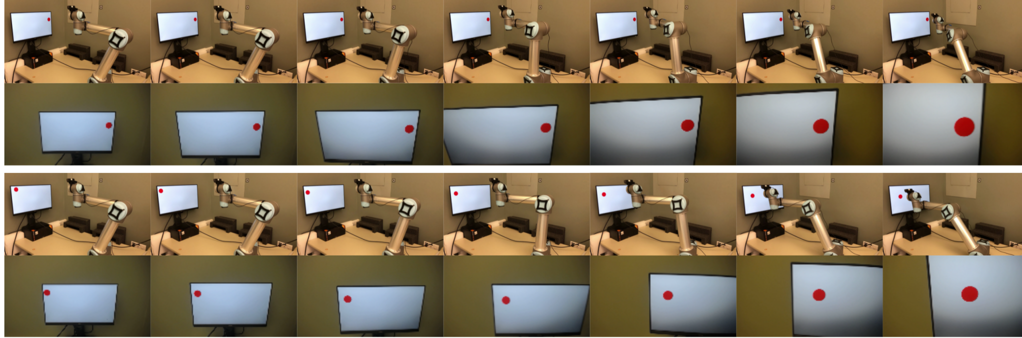

Figure 8.4: Learned behaviors in *Reaching* (first row) and *Tracking* (second row).

## 8.5 Learned Representation

In Figure 8.5, we show the coordinates captured by the spatial softmax layer as green marks. Though there are 32 coordinates in total because we have 32 output channels, some of the coordinates are overlapping with each other.

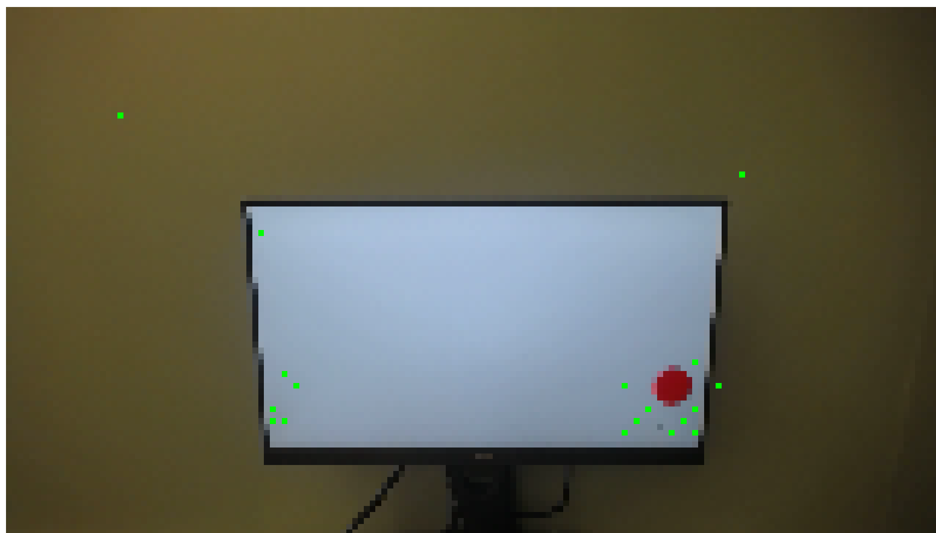

Figure 8.5: Coordinates captured by the spatial softmax layer.

# Chapter 9

# Conclusion

In the real world, time always marches on, and an agent interacts with the environment and learns from its experience in real-time. Unfortunately, this real-time property is not addressed in standard simulated environments. Consequently, most available open-source implementations of RL algorithms process computations sequentially, which is appropriate in simulated environments. However, when those implementations are deployed in the real world, whether they can maintain the performance becomes doubtful. One approach to solve this problem is through rearranging the computational orders, running environment interaction, and learning updates asynchronously.

In this thesis, we explored the challenge of real-time reinforcement learning, set up a vision-based robotic arm control environment, investigated how computational orders affect the performance of real-time systems, evaluated asynchronous reinforcement learning as an approach to address this problem, and provided potential improvement upon existing asynchronous learning architecture. In total, we ran 80 independent runs in our experiments, which took nearly 200 hours of usage on the robot. In most configurations, effective learning is achieved without tuning the hyper-parameters, which indicates the reliability of the algorithm, our learning architecture, and our task setup.

We found that the critical benefits of asynchronous learning are more flexibility in choosing action cycle time and better utilization of available computational resources. In sequential learning, the action cycle time is bounded by the time cost of learning updates. In contrast, in asynchronous learning,

the action cycle time is only constrained by the hardware capabilities. Unfortunately, given such flexibility, it is still unclear how to choose the action cycle time appropriately in a task-specific manner as either too long or too short cycle time could be detrimental to learning. Investigating the guidelines of choosing best-performing action cycle time in real-world reinforcement learning remains a promising future research direction.

When learning sequentially, the computational resource is constantly alternating between idle and busy mode, which causes waste in computation time. However, in asynchronous learning, especially based on our architecture, the gradient updates are being processed back to back so that the computation resources are fully utilized. However, when gradient update computation is cheap, asynchronous learning can also make gradient updates excessively faster than the environment interactions, which may lead to early convergence or overfitting. Devising approaches to avoid this issue while fully utilizing the computation resource is another promising research direction.

In conclusion, asynchronous reinforcement learning is a promising approach to dealing with real-time reinforcement learning challenges. However, it also has its intrinsic limitations. The decoupled environment interactions and learning updates require replay-buffer-based off-policy algorithms, which are usually both memory intensive and computationally expensive. Another approach researchers could resort to is developing more incremental and online update rules for learning agents to reduce per-step computations and memory usage and enable on-policy algorithms.

# References

Andrychowicz, O. M., Baker, B., Chociej, M., Jozefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., et al. (2020). Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, *39*(1), 3–20.

Barth-Maron, G., Hoffman, M. W., Budden, D., Dabney, W., Horgan, D., TB, D., Muldal, A., Heess, N., & Lillicrap, T. P. (2018). Distributed distributional deterministic policy gradients. *ICLR (Poster)*.

Bellman, R. (1958). Dynamic programming and stochastic control processes. *Information and control*, *1*(3), 228–239.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Coumans, E., & Bai, Y. (2016–2021). Pybullet, a python module for physics simulation for games, robotics and machine learning.

Dalal, G., Dvijotham, K., Vecerik, M., Hester, T., Paduraru, C., & Tassa, Y. (2018). Safe exploration in continuous action spaces. *arXiv preprint arXiv:1801.08757*.

Duan, Y., Chen, X., Houthooft, R., Schulman, J., & Abbeel, P. (2016). Benchmarking deep reinforcement learning for continuous control. *International conference on machine learning*, 1329–1338.

Dulac-Arnold, G., Levine, N., Mankowitz, D. J., Li, J., Paduraru, C., Gowal, S., & Hester, T. (2020). An empirical investigation of the challenges of real-world reinforcement learning. *arXiv preprint arXiv:2003.11881*.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., et al. (2018). Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *International Conference on Machine Learning*, 1407–1416.

Eysenbach, B., Gu, S., Ibarz, J., & Levine, S. (2018). Leave no trace: Learning to reset for safe and autonomous reinforcement learning. *ICLR (Poster)*.

Finn, C., Tan, X. Y., Duan, Y., Darrell, T., Levine, S., & Abbeel, P. (2016). Deep spatial autoencoders for visuomotor learning. *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 512–519.

Fujimoto, S., Hoof, H., & Meger, D. (2018). Addressing function approximation error in actor-critic methods. *International Conference on Machine Learning*, 1587–1596.

Gu, S., Holly, E., Lillicrap, T., & Levine, S. (2017). Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. *2017 IEEE international conference on robotics and automation (ICRA)*, 3389–3396.

Haarnoja, T., Ha, S., Zhou, A., Tan, J., Tucker, G., & Levine, S. (2018c). Learning to walk via deep reinforcement learning. *arXiv preprint arXiv:1812.11103*.

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018a). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *ICML*, *80*, 1856–1865.

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

Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., & Kavukcuoglu, K. (2017). Reinforcement learning with unsupervised auxiliary tasks. *ICLR*.

Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial intelligence*, *101*(1-2), 99–134.

Kalashnikov, D., Irpan, A., Pastor, P., Ibarz, J., Herzog, A., Jang, E., Quillen, D., Holly, E., Kalakrishnan, M., Vanhoucke, V., et al. (2018). Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. *arXiv preprint arXiv:1806.10293*.

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

Lange, S., Riedmiller, M., & Voigtländer, A. (2012). Autonomous reinforcement learning on raw visual input data in a real world application. *The 2012 international joint conference on neural networks (IJCNN)*, 1–8.

Laskin, M., Lee, K., Stooke, A., Pinto, L., Abbeel, P., & Srinivas, A. (2020). Reinforcement learning with augmented data. *NeurIPS*.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2016). Continuous control with deep reinforcement learning. *ICLR (Poster)*.

Mahmood, A. (2017). Incremental off-policy reinforcement learning algorithms.

Mahmood, A. R., Korenkevych, D., Komer, B. J., & Bergstra, J. (2018a). Setting up a reinforcement learning task with a real-world robot. *2018 IEEE/RSJ*

*International Conference on Intelligent Robots and Systems (IROS)*, 4635–4640.

Mahmood, A. R., Korenkevych, D., Vasan, G., Ma, W., & Bergstra, J. (2018b). Benchmarking reinforcement learning algorithms on real-world robots. *Conference on robot learning*, 561–591.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., De Maria, A., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., et al. (2015). Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*.

Peng, X. B., Andrychowicz, M., Zaremba, W., & Abbeel, P. (2018). Sim-to-real transfer of robotic control with dynamics randomization. *2018 IEEE international conference on robotics and automation (ICRA)*, 3803–3810.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015). Trust region policy optimization. *International conference on machine learning*, 1889–1897.

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

Schwab, D., Springenberg, J. T., Martins, M. F., Neunert, M., Lampe, T., Abdolmaleki, A., Hertweck, T., Hafner, R., Nori, F., & Riedmiller, M. A. (2019). Simultaneously learning vision and feature-based control policies for real-world ball-in-a-cup. *Robotics: Science and Systems*.

Stooke, A., Lee, K., Abbeel, P., & Laskin, M. (2021). Decoupling representation learning from reinforcement learning. *ICML*, *139*, 9870–9879.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. A Bradford Book.

Tassa, Y., Tunyasuvunakool, S., Muldal, A., Doron, Y., Trochim, P., Liu, S., Bohez, S., Merel, J., Erez, T., Lillicrap, T., et al. (2020). Dm_control: Software and tasks for continuous control. *arXiv preprint arXiv:2006.12983*.

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*. https://doi.org/10.3389/frobt.2018.00079.

Yahya, A., Li, A., Kalakrishnan, M., Chebotar, Y., & Levine, S. (2017). Collective robot reinforcement learning with distributed asynchronous guided policy search. *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 79–86.

Yarats, D., Kostrikov, I., & Fergus, R. (2021). Image augmentation is all you need: Regularizing deep reinforcement learning from pixels. *ICLR*.

Yarats, D., Zhang, A., Kostrikov, I., Amos, B., Pineau, J., & Fergus, R. (2019). Improving sample efficiency in model-free reinforcement learning from images. *arXiv preprint arXiv:1910.01741*.

Zhu, H., Yu, J., Gupta, A., Shah, D., Hartikainen, K., Singh, A., Kumar, V., & Levine, S. (2020). The ingredients of real world robotic reinforcement learning. *ICLR*.