Effective Real-time Reinforcement Learning for Vision-Based Robotic Tasks

by

Yan Wang

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

Vision is one of the essential means for humans to perceive the world. Similarly, today's intelligent robot agents rely on camera images to perform complex tasks in the real world. Due to the ever-changing nature of the real world, intelligent robot agents must continually learn from high-dimensional images to adapt to new environments. Such capabilities entail learning from images on-the-fly as they interact with the environments, which we call vision-based real-time learning.

Recently we have seen many successful applications of Reinforcement Learning (RL). It is natural to extend the scope of RL to vision-based real-time learning of robotic control tasks. However, a vision-based real-time robotic RL agent faces some practical issues oft-ignored in conventional RL research. The first issue is that robots deployed in the real world are usually tethered to a resource-limited computer, while vision-based RL algorithms are expensive. A prominent difference between real-time RL in the real world and conventional RL is that the time in the real world does not pause while the agent computes actions and updates policies. Given such a setup, it is unclear to what extent the performance of a learning system will be affected by resource limitations. Fortunately, in most cases, a powerful workstation can be wirelessly connected to the robot to provide extra computation resources. However, there is no systematic study of efficiently using the wirelessly connected powerful computer to compensate for performance loss. To shed some light on this issue, we propose and implement a real-time learning system called the Remote-Local Distributed (ReLoD) system to distribute computations of two deep reinforcement learning (RL) algorithms, Soft Actor-Critic (SAC) and Proximal Policy Optimization (PPO), between a local computer and a remote computer. The performance is evaluated on two vision-based robot tasks developed using a robotic arm and a mobile robot. Our results show that SAC's performance degrades heavily on a resource-limited computer. Strikingly, distributing all computations of SAC on a wirelessly connected workstation fails to improve performance. However, a carefully chosen distribution consistently and substantially improves performance on both tasks. On the other hand, the performance of PPO remains largely unaffected by the distribution of computations. In other words, without careful consideration, using a powerful remote computer may not improve performance.

The second issue a real-time robotic RL agent faces is that designing dense rewards for vision-based real-robot tasks requires hand-engineering or pre-training, which can be unsuitable for unforeseen tasks. When formulating a real-world robotic task as a reinforcement learning (RL) task, it is crucial to determine a reward function that is convenient to specify, accurately captures the intended problem, and facilitates the agent with learning. Many designers of real-world robot tasks use domain knowledge to design informative dense rewards to facilitate training. However, designing task-dependent reward functions for real-time learning tasks, including non-vision- and vision-based tasks, are difficult since the domain knowledge is generally unavailable for non-stationary and unforeseen environments. Moreover, hand-crafting a dense reward function for vision-based tasks is more problematic due to the need for effective image encoders, which are generally unavailable prior. For so-called goal-reaching tasks, there is a simple way of designing the reward function independent of domain knowledge and prior image encoders but still aligning well with our intention: giving a -1 reward every time step. Goal-reaching tasks are formulated as episodic RL tasks with termination upon reaching the terminal state as soon as possible. We call them minimum-time tasks or vision-based minimum-time tasks if images represent terminal states since maximizing the undiscounted sum of these -1s leads to reaching the terminal state as soon as possible. Unfortunately, minimum-time tasks are usually avoided in practice, as they are considered difficult and uninformative for learning. In this thesis, we demonstrate that non-vision and vision-based minimum-time tasks can be learned quickly from scratch. We also provide guidelines that practitioners can use to predict if the minimum-time task formulation is appropriate for their problems based on the performance of the initial policy. Following our guidelines on minimum-time tasks, we first demonstrate using a single reinforcement learning system to achieve real-time learning of pixel-based control for several different kinds of real robots from scratch.

Preface

The first part of the work (Chapter 4 and Chapter 5) was accepted at the 2023 International Conference on Robotics and Automation (ICRA). It was inspired by Yufeng Yuan's paper (Yuan & Mahmood, 2020). Our work originated when Professor Rupam Mahmood realized that components of a learning system could be distributed between two computers to allow robots with limited onboard computation resources to leverage expensive state-of-the-art reinforcement learning algorithms. For this purpose, I developed the Remote-Local Distributed (ReLoD) system.

In a meeting with Rupam, we realized that hand-crafting a dense reward function is difficult for vision-based tasks. This discussion inspired the second part of this work: to use a reward function of -1 for each step as an alternative reward function for tasks that try to reach terminal states as soon as possible. This part of the work (Chapter 6 to Chapter 8) was submitted to the 2023 International Conference on Intelligent Robots and Systems (IROS). The SAC learning system used in this work was initially developed by Yufeng (Yuan & Mahmood, 2020), and I extended it to support the ReLoD system we proposed. The UR5-VisualReacher task was also initially developed by Yufeng (Yuan & Mahmood, 2020), and I modified its reward function to -1 for each step. We call the modified task UR5-VisualReacher-MinTime. I also developed the Create-Reacher task and the Reacher simulation task.

The first part of this work (Chapter 4 and Chapter 5) was done in cooperation with Gautham Vasan and Rupam Mahmood. The second part of this work (Chapter 6 to Chapter 8) was done in cooperation with Gautham Vasan, Fahim Shahriar, and Rupam Mahmood. Gautham developed the tasks Vector-ChargerDetector, Ball-in-Cup, and Dot-Reacher in Chapter 6 and Chapter 8.

He also implemented the learning system used in the second part of this work to solve simulation tasks. He ran experiments for Vector-ChargerDetector in the real world and Dot-Reacher in simulation. In addition, he extended SpinUp PPO (Achiam, 2018) to support the ReLoD system. Fahim Shahriar developed the task Franka-VisualReacher in Chapter 8 and ran experiments for it in the real world. I performed the Create-Reacher experiments, the UR5-VisualReacher experiments, the UR5-VisualReacher-MinTime experiments, the Ball-in-Cup simulation experiments, and the Reacher simulation experiments for this work. Gautham, Rupam, and I discussed and finalized the experiment setups in this thesis. We also wrote, edited, and submitted the two above papers about this work.

Acknowledgements

I am indebted to my academic advisor Rupam Mahmood for offering me an opportunity to grow professionally. During my study, I have learned much from him and benefited greatly from his knowledge, patience, and meticulous editing. His levels of knowledge and ingenuity are something I will keep on pursuing. I am extremely appreciative that Rupam took me on as his master's student and continued to have faith in me over the years. This thesis would not have been possible without his support, feedback, and guidance.

I also must thank my collaborator, Gautham Vasan, and Fahim Shahriar, who investigated a tremendous amount of time on the experiments. Their encouragement, patience, and cooperation make this thesis possible. Further, I would like to give special thanks to Yufeng Yuan, whose extraordinary work on real-time learning inspired and laid the foundation for this thesis.

I acknowledge funding from the CCAI Chairs program, the RLAI laboratory, Amii, NSERC of Canada, and the donation of the UR5 arm from the Ocado Group. In addition, I am grateful to have access to Compute Canada, which reduced the time to run experiments from months to weeks.

Many thanks to the teachers at the computing science department who taught and helped me during my study at the University of Alberta. They are but are not limited to: Pierre Boulanger, Lili Mou, Ioanis Nikolaidis, Jonathan Schaeffer, Rich Sutton, Martha White, Levi Lelis, and Osmar Zaiane.

Last but not least, I want to express my deepest gratitude to all my friends who helped me. A non-exhaustive list is: Amir Karimi, Blanca Miller, Fengdi Che, Homayoon Farrahi, Jiamin He, Mohamed Elsayed, Qingfeng Lan, Samuele Tosatto, Shibhansh Dohare, and Shivam Garg.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Building intelligent robotic agents capable of learning useful skills from real-world interactions is one of the long-standing goals of embodied artificial intelligence. Vision is a powerful sensory modality for extracting information about the real world. Therefore, intelligent robotic agents are generally expected to learn from camera images. Indeed, today's real-world robots rely on images taken from cameras to perform complex tasks. For example, an autopilot system relies on the critical information provided by camera images, such as the locations of pedestrians and other vehicles, to generate proper driving commands. Furthermore, intelligent robotic agents who perform in the real world also need to adapt to ever-changing or unseen environments, as the real world is intrinsically non-stationary. As a result, the adaptive capability entails vision-based learning on the fly as the agent interacts with the physical world, also known as **real-time learning of vision-based tasks**. However, classic control methods are often less effective in non-stationary environments. Thanks to the progress of deep neural networks, Reinforcement Learning (RL) (Sutton & Barto., 2018) has been successfully applied to many fields, including robotic control tasks. Researchers have demonstrated many remarkable works in simulation (Berner et al., 2019; Li et al., 2020; Mnih et al., 2016) and in the real world (Levine et al., 2016; Tian et al., 2020; Tzeng et al., 2020). Thus, real-time RL is promising for vision-based robotic learning tasks under unseen or changing environments.

But extending conventional RL methods to vision-based real-time learning is difficult due to the discrepancy between simulation and the real world. For example, when learning in real-time, the real world does not pause while the agent computes actions or makes learning updates (Mahmood et al., 2018a; Ramstedt et al., 2019). Moreover, the agent obtains sensorimotor information from various onboard devices and executes action commands at a specific frequency. Given these constraints, a real-time learning agent must compute an action within a chosen action cycle time and perform learning updates without disrupting the periodic execution of actions (Yuan & Mahmood, 2022). Unfortunately, there has been relatively little interest in studying real-time learning in the real world.

1.1 Problem Statement

As a step toward robust real-time learning of vision-based robotic tasks in the real world with RL, we aim to investigate two main hurdles in this thesis:

- Most real-time robotic applications have a resource-limited computer available locally. But vision-based deep RL is expensive. This is a significant reason why real-time RL is difficult for vision-based robot tasks.
- Current deep RL tasks are mainly based on dense rewards, which require hand-crafted functions such as calculating Euclidean distance between the current and the terminal state or shaping through a procedure. Reliance on such an expensive and domain-knowledge-dependent method impedes the application of RL to unseen tasks. Besides, designing a dense reward function for vision-based tasks is generally difficult due to the lack of effective image encoders.

Details about the two hurdles are described in the following sections.

1.2 Limited Onboard Computations

State-of-the-art RL algorithms and image encoders are computationally intensive, and hence for real-time vision-based robotic control, they go together with a computationally powerful computer tethered to the robot (Yuan & Mahmood, 2022). However, a robot agent deployed in the real world typically uses a resource-limited tethered computer, while a powerful workstation is often wirelessly connected to the robot (Haarnoja et al., 2019; Bloesch et al., 2022). In this thesis, we use **local** to refer to the computer tethered to the robot and **remote** to refer to the wirelessly connected computer.

Unlike learning in the simulation, where the agent performance is measured against environmental steps, we should measure the performance against wall clock time in real-time learning. That is because training on real robots is expensive and may be dangerous. We desire a learning system to solve the given task as fast as possible. Since resource-limited computers need more wall clock time to compute actions and update policies, the policy update frequency will be reduced on resource-limited computers. On the other hand, the policy update frequency will be unaffected in simulation environments because simulators used in RL research often pause the time march while the agent computes actions and updates policies. In this thesis, we evaluate the performance against wall clock time instead of total environmental interactions.

It is unclear how much resource limitations impact the performance of a learning system due to the reduced policy update frequency. Moreover, computations of a vision-based learning system using these two computers can be distributed in different ways. Unfortunately, prior works do not systematically study distributions of computations between local and remote computers nor suggest how to achieve an effective distribution to compensate for the performance loss. [1]

To better understand this problem, we develop two vision-based tasks using a robotic arm and a mobile robot and propose a real-time RL system called the Remote-Local Distributed (ReLoD) system. Similar to Yuan and Mahmood's (2022) work, ReLoD parallelizes computations of RL algorithms to maintain short action-cycle times and reduce the computational overhead of real-time learning. But unlike the prior work, it is designed to utilize both a local and

[1]The study of this problem resulted in a paper accepted at the 2023 International Conference on Robotics and Automation (ICRA).

a remote computer. ReLoD supports three different modes of distribution: Remote-Only which allocates all computations on the remote computer, Local-Only which allocates all computations on the local computer, and Remote-Local which carefully distributes the computations between the two computers in a specific way.

Our results show that the performance of SAC on a tethered resource-limited computer drops substantially compared to its performance on a powerful workstation. Surprisingly, when all computations of SAC are deployed on a wirelessly connected powerful workstation, the performance **does not** improve notably. This observation contradicts our intuition as this mode fully utilizes the resources of a workstation. On the other hand, SAC's Remote-Local mode consistently improves its performance by a large margin on both tasks, which indicates that a careful distribution of computations is essential to utilize a powerful remote workstation. However, the Local-Remote mode only benefits computationally expensive and sample-efficient methods like SAC since the relatively simpler learning algorithm PPO performs similarly in all three modes. We also notice that the highest average return attained by PPO is about one-third of the highest average return attained by SAC, which indicates that SAC is more effective in complex robotic control tasks.

The ReLoD system in the Local-Only mode can achieve a performance that is on par with a system well-tuned for a single computer (Yuan & Mahmood 2022), though the latter overall learns slightly faster. This property makes our system suitable for conventional RL studies as well.

1.3 Difficulty of Designing Dense Rewards

In reinforcement learning, the task designer implicitly specifies the desired behavior using the reward function. Task designers often rely on task-specific domain knowledge to hand-craft a dense reward signal with state-to-state differences that facilitate faster learning and guide the agent to a reasonable solution. For example, the reward function of the Reacher environment from OpenAI Gym (Brockman et al., 2016) depends on the Euclidean distance of the fingertip of a robot arm from a target. In this thesis, we call these guiding rewards as they go beyond specifying what task to solve and guide the agent on how to solve the task.

Although guiding rewards have many advantages, they need to be revised in real-time vision-based learning. First, since a guiding reward function depends on the environment's domain knowledge, it is difficult, if not impossible, to design a guiding reward function capable of capturing the ever-changing characteristics of a non-stationary environment. Besides, an intelligent agent cannot rely on a human instructor to provide domain knowledge beforehand to solve unforeseen tasks. Second, hand-crafting a guiding reward function based on images is often tedious and error-prone, as effective image encoders for extracting useful information from pixels are usually unavailable prior. Third, guiding rewards often bias the learned control policy in a potentially sub-optimal way. Since hand-crafted rewards reflect the task designer's domain knowledge and preferred behaviors, it could bias the solutions that the agent can find (Riedmiller et al., 2018). Thus, in the vision-based real-time learning setting, it is beneficial to work with reward signals independent of domain knowledge and prior image encoders without biasing the solution in undesirable ways.

For most goal-reaching problems, there is a simpler alternative to guiding rewards that is easy to specify but still incorporates our intended goal accurately. If we consider goal-reaching problems as episodic tasks with termination upon reaching the terminal state, the reward can be simply a constant negative scalar every time step. We call them **minimum-time** tasks, and they can be seen as a special case of sparse-reward tasks. Similarly, if tasks are vision-based and terminal states are represented as images, we call them **vision-based minimum-time** tasks. Using the minimum-time formulation, the task designer can easily avoid the aforementioned problems by focusing solely on recognizing task success rather than trying to guide the learned policy with domain knowledge. This specification is also simpler than the standard sparse rewards, where a positive discounted reward is only given when the terminal state is reached because minimum-time tasks can be undiscounted.

Unfortunately, sparse-reward tasks, including minimum-time tasks, are generally hard to solve, not to mention the vision-based minimum-time tasks. With guiding rewards, we get state-by-state reward differences, which can be informative for policy improvement. In addition, with guiding rewards, there are early signs of learning so that we can quickly determine whether the learning is happening. On the other hand, sparse-reward tasks may take much longer to show any sign of learning as informative signals are given only sparsely.

In this thesis, we take the first step towards efficiently designing a reward function for real-time learning of vision-based robot tasks. More specifically, we propose guidelines to determine if a minimum-time task is solvable by the state-of-the-art RL algorithm SAC before the actual training. We identify that SAC can reliably solve complex vision-based tasks in the minimum-time formulation if the agent can reach the terminal states represented by an image often enough using its initial policy. In this work, the initial policy is the Gaussian distribution $\mathcal{N}(0, 1)$ for all tasks, including visual and non-visual tasks. We empirically demonstrate that the performance of the initial policy can be used to predict whether a minimum-time task, including non-vision and vision-based tasks, can be learned quickly and reliably using SAC. We establish that the time limit should be treated as a tunable solution parameter that an agent can tweak rather than a part of the problem specification. Practitioners can utilize our proposed guidelines to determine if a minimum-time task formulation is appropriate for their problems. Following our guidelines on minimum-time tasks, we produce the first demonstration of real-time learning of pixel-based control for several kinds of physical robots from scratch.

1.4 Contributions

The main contributions of the first part of this thesis include the following: [2]

- We developed and publicized a system called ReLoD to distribute com-

[2]The first part of this thesis was accepted at the 2023 International Conference on Robotics and Automation (ICRA).

putations of an RL algorithm between a local computer and a remote computer. As far as we know, ReLoD is the first publicly available system for real-time RL that applies to multiple robots for vision-based tasks. The source code can be found at `https://github.com/rlai-lab/relod`

- We developed three simulated tasks and four real robotic tasks that can be used as benchmarking tasks for real-time learning of minimum-time tasks. The four real robotic tasks are all vision-based and developed with four different physical robots to provide diversity.

- We used the ReLoD system and a vision-based task to systematically study how much limited onboard computations impact the performance of SAC.

- We used the ReLoD system and two vision-based tasks to systematically investigate how best to distribute computations of SAC and PPO between a resource-limited local computer and a powerful remote computer to compensate for the performance loss.

The main contributions of the second part of this thesis include the following:

- We formulated minimum-time tasks and demonstrated that, under certain conditions, SAC could effectively solve real-time non-vision and vision-based minimum-time tasks in both simulation and the real world. The real-world demonstration uses the four vision-based minimum-time robotic tasks mentioned above. As far as we know, this is the first demonstration of the feasibility of using a single system to solve multiple real-time vision-based minimum-time robotic tasks across several different robots in the real world.

- We proposed and tested guidelines to determine whether SAC can effectively solve a real-time learning problem in the minimum-time setting based on the performance of the initial policy.

Chapter 2

Literature Review

This chapter briefly discusses different approaches to vision-based robotic control tasks, from classic control to real-time RL.

2.1 Visual Seroving

Conventional non-vision robot controllers suffer control errors induced by uncertainties in robot models and environments (Weiss et al., 1987). To reduce the control error, researchers explored using images to compensate for the inaccurate robot models and environments. Such methods are called visual seroving control.

Weiss et al. (1987) proposed the design of an adaptive image-based visual servo controller (IBVS) where image feedback was used in the dynamic closed-loop control. They evaluated the proposed controller's performance on one-, two-, three-, and five-DOF systems in simulation. Although their simulated results show that the adaptive image-based visual servo controller can improve control accuracy and speed compared to the traditional position-based controller, deploying an adaptive image-based visual servo controller in the real world is still challenging. One will have to overcome practical issues such as the computation of the image Jacobian matrix for real tasks, camera calibration, and image processing delay.

Koivo et al. (1991) showed the real-world use of adaptive image-based visual servo controllers to grasp a moving object with an industrial robotic arm. However, their methods assume the mapping between the image coordinates and the world coordinates is known, which requires careful calibration of cameras. As a result, cameras must be fixed in known locations, limiting the real-world deployment of their method.

Conkie et al. (1990) focused on an approach that does not require careful calibration of cameras since their method measures the displacement from the object to the target in the image coordinates. It estimates the image Jacobian matrix by running a few extra small movements of each joint that are not meant to solve tasks and recording the changes of a reference image feature in the image coordinates. A similar approach was also described by Yoshimi et al. (1994). The main limitation of their approaches is that they assume the image Jacobian matrix is smooth and flat, which may not hold in real-world tasks.

Martin Jägersand (1996) proposed a trust region method to continuously update the image Jacobian matrix along a trajectory. Unlike previous works, it uses an MSE-alike update rule to enable estimating from arbitrary movements. As a result, it does not need extra movements. It does not strictly assume the image Jacobian matrix is smooth and flat since the trust region optimization enforces the validity of the current estimation.

Although visual seroving methods are effective in robot control tasks, they share some common problems. First, they require domain knowledge to design an image encoder to extract image features manually. As the image encoder is not learned, it may not work in other tasks nor adapt to changing environments. Moreover, visual seroving methods require a predefined target, and the target has to be inside the camera view all the time, which is not always possible in real-world applications. More difficulties of IBVS, such as singularities of the image Jacobian matrix, were discussed in the paper by Chaumette et al. (1998). Although researchers proposed new methods (Malis et al., 2001, Corke et al., 2001, and Collewet et al., 2002) to handle the problems of IBVS, they still use fixed image encoders and predefined targets. In addition, some new methods also require offline procedures such as 3D models of targets to estimate the depths of image features. In conclusion, since visual seroving methods use domain knowledge and offline techniques to design effective controllers, they are unsuitable for real-time learning in ever-changing environments.

2.2 Supervised Learning from Demonstration

Learning from Demonstration refers to the methods that teach robots new skills by imitating expert behaviors (Ravichandar et al., 2020). It has many categories. In this thesis, we only focus on vision-based Learning from Demonstration with machine learning methods.

Pomerleau first developed ALVINN (Autonomous Land Vehicle In a Neural Network) for the task of road-following on a test vehicle. The network takes camera images and a laser range finder as the inputs, and produces directions to follow as the output. They used backpropagation to train the network on simulated image data. Bojarski et al. (2016) described an end-to-end approach to learning from human demonstration for autonomous driving on roads. They trained a Convolutional Neural Network (CNN) to map camera images to steering commands. To increase the robustness of the learned policy, they augmented the training images with three cameras facing at different angles. The weights of the CNN are adjusted to minimize the mean square errors between the outputs and the sheering commands provided by a human driver. One major issue with this method is that the policy learned in this way does not consider the passenger's intention. To address this problem, Codevilla et al. (2018) proposed *Conditional Imitation Learning* in a more recent paper. In conditional imitation learning, the driver's intentions are recorded in the demonstration and fed into the neural network in training. Similarly, the policy is optimized by minimizing the mean square errors between the network outputs and the driver's commands. All those methods are examples of the simplest form of learning from demonstration, called **direct behavior cloning**, as they try to learn actions provided by experts directly.

However, direct behavior cloning has yet to be widely used in robotic arm manipulation tasks due to visual artifacts such as human hands in demonstration images (Young et al., 2020). Zhang et al. (2018) proposed a VR system to collect demonstration data for robotic arm manipulation tasks to minimize visual artifacts. This system ensures that humans and robots share the same observation and action spaces. They showed that direct behavior cloning could be used to solve complex robotic arm manipulation tasks, including grasping and placing. Sharma et al. (2019) proposed a decoupled hierarchical controller to allow robots to learn robotic arm manipulation tasks from human demonstration from the third-person view. In their methods, the observation spaces of humans and robots are different. They trained two levels of modules independently to separate what to do and how to do the task. The higher-level module is a goal generator that takes as input the human demonstration images in the third-person view and the robot demonstration images in the first-person view. The goal generator is trained to output image goals in the first-person view k frames later. The lower-level controller takes as input the goal images and the robot demonstration images in the first-person view. It is trained with kinesthetic teaching to output actions to achieve the generated goal images. They showed that the learned goal generator and the controller could be generalized to unseen objects and tasks.

Another problem of direct behavior cloning is the distribution mismatch between demonstration and training. Researchers have proposed many methods to correct distribution mismatch. Ross et al. (2011) proposed a dataset aggregation method called **DAgger**. DAgger is a human-in-the-loop method that asks an expert to label actions for states visited under the training distribution. The policy is optimized to match expert actions. It can be proven that DAgger can significantly reduce distribution mismatch. Although DAgger is simple, it is not generalizable to robotic applications in the real world as it assumes the availability of expert-labeled actions. Pervez et al. (2017) proposed **Deep Dynamic Movement Primitives (D-DMP)** to increase generalization without asking for more expert-labeled actions. They used expert data to train a DMP controller for a robotic arm to perform object manipulation tasks. A CNN was trained offline to predict the locations of various objects and targets for the DMP controller with ground truth. They claimed that the trained policy could be generalized to unseen objects and targets.

Although learning from demonstration has been successfully applied to many fields, such as assembly line operations (Zhu et al., 2018, Vogt et al., 2017), physical rehabilitation (Vasan et al., 2017), and UAV control (Ross et al., 2013), it has major problems limiting their application. First, it is not always intuitive for humans to provide demonstrations to robots. For example, providing demonstration data to humanoid robots is difficult due to the fundamental difference in kinematics. Second, it relies on the availability of human demonstration. Third, it assumes that expert data are optimal, which often does not hold in reality. Therefore, learning from demonstration is not suitable for the real-time learning setting.

2.3 RL for Robot Control

Recently RL methods has been applied by researchers to solve complex robot control tasks. As one of the earliest works, Kohl et al. (2004) proposed a 'policy gradient method' to train an Aibo robot to walk forward as fast as possible. Their method aims to find the best parameters for higher-level control. Due to the problem formulation, their proposed method differs significantly from today's policy gradient approach and only applies to other robots and tasks.

2.3.1 Sim-to-Real Methods

Since collecting samples in the real world is expensive, researchers developed many techniques to accelerate the training. For example, sim-to-real methods collect samples and train agents in high-fidelity simulators and transfer learned policies to real robots.

Krishnan et al. (2019) introduced an open-source simulator and a gym environment for quadrotors to train agents in the simulator. Since onboard computing is scarce and updating RL policies with existing methods is computationally intensive, they carefully designed policies considering the power and computational resources available onboard.

Zhu et al. (2017) introduced a 3D simulator called AI2-THOR framework to generate high-quality 3D scenes to train agents to reach specific targets.

Then the trained policy was deployed to the real robot with fine-tuning training to achieve the same task in similar real-world scenes. James et al. (2016) also developed a 3D simulator to train a robotic arm to lift a cube. The simulator was carefully crafted to resemble the real-world setup They showed that the trained policy could be directly transferred to the real robot. However, these two works require high-fidelity simulators to minimize the discrepancy between simulation and the real world.

More advanced methods were proposed to mitigate the inaccuracy of simulation. For example, Tzeng et al. (2015) suggested a novel method to align simulated and real-world observations. In this method, two additional losses, domain confusion loss and pairwise loss, were added to the task loss to allow a more robust transfer of learned policies from simulation to the real world. Another work (Rusu et al., 2016) used progressive networks to close the gap between simulation and the real world. They first train a policy in a simulator and then train the second policy in the real world using the outputs of the first policy as the inputs.

Although sim-to-real methods are impressive in solving complex tasks, they do not apply to real-time learning as they need prior knowledge to model robots and tasks in simulation.

2.3.2 Offline Learning Methods

Another popular way to tackle difficulties in collecting samples in the real world is to use offline RL methods. In offline RL, agents learn policies exclusively on previously collected data without any online interactions with the environments. Offline RL is appealing for robotic learning tasks since agents can learn new skills from previously collected data.

Pinto and Gupta (2016) trained a CNN model with offline data to predict angles for grasping. They showed that model could have better generalization with more data. Levine et al. (2018) proposed a framework to collect data on a large scale from multiple robots and used offline dynamic programming to train a CNN to predict the outcome of a grasping action. They also showed that data collected from other robots could improve networks' generalization ability, which revealed the value of offline RL. Kalashnikov et al. (2018) also demonstrated a similar approach.

Besides grasping tasks, Finn et al. (2016) presented a model-based offline RL approach based on the deep dynamics approach introduced by Wahlström et al. (2015) to train a spatial autoencoder and a simple Linear-Gaussian controller to perform various tasks on real robots. This method pretrains a non-visual controller to facilitate effective exploration with offline data. Later the same author introduced the visual foresight method for motion planning (Finn et al., 2017). They trained a video-prediction model offline that predicts the possible consequence of actions in the image space from a batch of stored videos. Motion is then planned on the learned video-prediction model to move an object to a destination. Followup works (Ebert et al., 2018; Dasari et al., 2019; Tian et al., 2021) showed the effectiveness of the visual foresight method in solving diverse tasks. In the navigation field, Mo (2018) presented a dataset of real-world scenes for offline training of navigation systems, and Kahn et al. (2020) demonstrated the effectiveness of offline RL in learning navigation policies from data collected with random exploration.

Although offline RL methods are very effective in training robot controllers, they are not suitable for real-time learning for two reasons. The first reason is that offline RL does not allow online interaction with environments once data are collected, which contradicts the requirement of real-time learning that agents must adapt to ever-changing environments. Another reason is that all offline RL methods suffer from the same intrinsic problem: distribution shift. Although many methods, such as importance sampling correction (Liu et al., 2018; Zhang et al., 2020a; Zhang et al., 2020b), policy constraint (Fujimoto et al., 2019; Kumar et al., 2019), and conservative Q learning (Kumar et al., 2020), have been proposed to mitigate distribution shift, their effectiveness has not been tested on real robots.

2.4 Real-time Reinforcement Learning

This section describes works exploring real-time learning of robot tasks with reinforcement learning methods. Real-time learning means agents continually learn on the fly as they interact with environments. However, there are surprisingly few works that focus on real-time learning.

One early work from Mahmood et al. (2018a) proposed six benchmarking tasks developed with three physical robots for real-time learning in the real world. They compared the performance of four reinforcement learning algorithms on the six tasks and showed that real-time learning of robot tasks is possible if the learning system is carefully set up. They also showed that delays in communicating actions over WiFi could significantly worsen an agent's performance. The desire to minimize WiFi latency inspired this work.

Later Haarnoja et al. (2019) showed the feasibility of real-time learning of robot tasks with distributed reinforcement learning systems. They proposed a parallel learning system tailored to learn a stable gait using SAC and the minitaur robot (Kenneally et al., 2016). Their system collects samples with the onboard computer and updates policies with a powerful wirelessly connected computer. The policies are periodically synchronized between the two computers. A recent paper by Smith et al. (2022) demonstrated real-time learning of walking gait from scratch on a Unitree A1 quadrupedal robot on various terrains with model-free methods. Their learning system is sequential and runs solely on the onboard computer, which requires many resources. The above works focus on non-vision tasks and do not address real-time learning issues. Instead, we focus on vision-based tasks in this work, which is significantly harder, and address two important real-time learning issues.

A system comparable to ReLoD is SenseAct, which provides a computational framework for real-time robotic learning experiments to be reproducible in different locations and under diverse conditions (Mahmood et al., 2018b). This paper shows that many factors such as action space definition, communication latency, order and concurrency of computations, and action cycle time have significant impacts on the performance of real-time learning tasks. Although SenseAct enables the systematic design of real-time robotic tasks for RL, it does not address how to distribute computations of a real-time learning agent between two computers, and the original work does not contain vision-based tasks. But we use the guiding principles of SenseAct to design vision-based robotic tasks and systematically study the effectiveness of different distributions of computations of a learning agent.

The work from Yuan and Mahmood (2022) investigated the policy update delay problem in the real-time learning setting. Unlike the previous works in this section, this work uses vision-based robotic tasks. Based on their results, a short action cycle time could not be properly maintained in sequential RL learning systems as the policy updates would take a long time to complete. They proposed to use a separate process to perform expensive policy updates (asynchronous learning) so that the agent-environment interaction would not be blocked. Note that asynchronous learning in their work and this thesis does not mean using multiple environment instances to improve sample collection efficiency. Instead, it means using concurrency to parallelize the computations of a learning system to reduce computational overhead. Their results showed that an asynchronous learning system could effectively maintain short action cycle times. Their work focuses on efficiently parallelizing computations of a learning system on a single computer given enough resources.

This thesis is the first one that combines Yuan's work (2022) and Mahmood's work (2018a, 2018b) since we used vision-based robotic tasks to systematically study how to set up a real-time learning system using two computers. More especially, we extended their work by evaluating the performance loss of a real-time learning system on a resource-limited computer and investigating how to compensate for the performance loss with a wirelessly connected powerful computer.

2.5 Distributed Reinforcement Learning

This section describes works that run RL learning systems in a distributed manner on different computers to allow more effective learning.

Nair et al. (2015) proposed a distributed learning architecture called the GORILA framework that mainly focuses on using multiple actors and learners to collect data in parallel and accelerate training in simulation using clusters of CPUs and GPUs. GORILA is conceptually akin to the DistBelief (Dean et al., 2012) architecture. In contrast to the GORILA framework, our ReLoD system focuses primarily on how best to distribute the computations of a learning system between a resource-limited local computer and a powerful remote computer to enable effective real-time learning. In addition, the GORILA framework is customized to Deep Q-Networks (DQN), while our ReLoD system supports two policy gradient algorithms using a common agent interface.

A work similar to GORILA is the asynchronous methods proposed by Mnih et al. (2016). Their asynchronous methods, such as A3C and asynchronous Q learning, use multiple actors and environment instances to accelerate sample collection and decorrelate samples. Actors periodically update their policies with a globally shared set of parameters. Since actors run in parallel, their policies are updated in an asynchronous manner. However, like the GORILA framework, their asynchronous methods do not focus on how best to distribute the computations of a learning system between a resource-limited local computer and a powerful remote computer to enable effective real-time learning. Unlike our ReLoD system, their asynchronous methods run all components on a single computer.

Lambert et al. (2019) used a model-based reinforcement learning approach for high-frequency control of a small quadcopter. Their proposed system computes actions and updates policies on a remote computer. Bloesch et al. (2021) used a distributed version of Maximum aposteriori Policy Optimization (MPO) (Abdolmaleki et al., 2018) to learn a vision-based control policy that can walk with Robotis OP3 bipedal robots. The robot's onboard computer samples actions and periodically synchronizes the policy's neural network weights with a remote learning process at the start of each episode. Those papers aim at solving tasks instead of systematically comparing different distributions of computations of a learning agent between a resource-limited computer and a powerful computer. In addition, their systems are tailored to specific tasks and algorithms and are not publicly available, while ReLoD is open-source, task-agnostic, and compatible with multiple algorithms.

2.6 Sparse Reward

This section describes works that aim at solving sparse reward problems. The main idea is to improve exploration by providing external reward signals that are not part of the problem.

Riedmiller et al. (2018) proposed SAC-X, a method capable of learning from sparse rewards. It simultaneously learns policies on a set of auxiliary tasks by actively scheduling and executing those tasks to explore its observation space in search of sparse rewards of externally defined target tasks. Hertweck et al. (2020) also focused on learning from sparse rewards. To facilitate effective exploration, they propose using agent-internal auxiliary tasks. Their work can be viewed as an extension of Riedmiller et al. (2018), where they proposed a way to define auxiliary tasks which can be integrated with SAC-X to solve complex tasks like Ball-in-Cup from scratch. Only raw sensor streams were used for controller inputs and the auxiliary reward definition. However, the choice of auxiliary tasks depends on domain knowledge and is not transferable to other tasks or robots, making their methods not suitable for real-time learning tasks.

Andrychowicz et al. (2017) proposed Hindsight Experience Replay (HER), a technique that enables an agent to learn from failed episodes by treating the previously seen states as a pseudo-goal. It can be seen as an exploration strategy that helps the agent learn from sparse reward feedback. However, it only applies to environments where every reachable state can be considered a terminal state. If the task specification involves only one or a small set of terminal states, HER will not help improve the sample efficiency of the agent. In addition, it cannot be directly extended to all vision-based tasks where the presence of distinct visual features often characterizes terminal states. While Nair et al. (2018) extended HER to solve vision-based tasks like Reaching and Pushing using a 7-DoF Sawyer arm, their approach relies heavily on a setup involving a fixed arena with an overhead camera that can always view the target. It is also only applicable to environments where every single reachable state on a 2D plane can be considered a terminal state. As a result, HER-based methods are not suitable for real-time learning tasks.

All these works propose methods to improve exploration in sparse reward tasks and thus facilitate fast learning. Our work does not focus on novel strategies for exploration. Instead, we propose guidelines that help to predict whether a minimum-time task can be learned quickly and reliably using SAC based on the performance of the initial policy.

Chapter 3

Background

In this chapter, we first discuss the components of the Reinforcement Learning framework and how an RL problem is modeled with the Markov Decision Process. Then we introduce two state-of-the-art RL algorithms, Soft Actor-Critic and Proximal Policy Optimization, used in this thesis to solve vision-based real-time learning tasks. As most of our tasks are image-based, we also provide details about our image-processing architecture, including the network design, image representation, and data augmentation. Finally, we briefly describe the SenseAct framework, which enables a unified and systematic design of our real robot tasks.

3.1 Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning whose goal is to learn the optimal behavior from trial and error to obtain the maximum long-term numerical return (Sutton & Barto, 2018). A typical RL system has two major components: the learning agent and the environment. In the conventional RL setting, the agent and the environment interact at discrete timesteps, as shown in Fig. 3.1.

The agent-environment interaction is often modeled with a Markov Decision Process (MDP). An MDP can be characterized by a tuple with 1) the state space $\mathcal{S}$, 2) the action space $\mathcal{A}$, 3) the initial state distribution μ: 4) the transition dynamics $\mathcal{P}$: 5) the reward function $\mathcal{R}$, and 6) the discount factor γ. That is, an MDP is represented as $M = (\mathcal{S}, \mathcal{A}, \mu, \mathcal{P}, \mathcal{R}, \gamma)$. At the beginning of

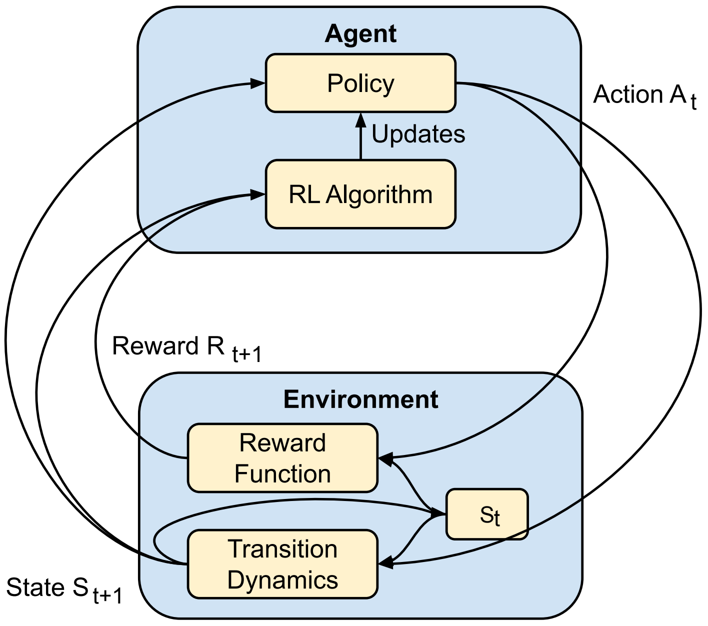


Figure 3.1: The agent-environment interaction loop in reinforcement learning

the agent-environment interaction, the environment is reset to an initial state by sampling from the initial state distribution: $s_0 \sim \mu$. For each timestep t, the agent uses a function called **policy** $\pi : \mathcal{S}^t \rightarrow \Delta(A)$ to map the transition history to an action distribution $\pi(\cdot|S_0, S_1, ..., S_t)$. Here, the notation $\Delta(\mathcal{A})$ denotes the set of all probability distributions over the action space $\mathcal{A}$. Since the entire transition history is hard to handle in practice, we further assume that MDPs satisfy the **Markov Property**, which says:

$$\pi(\cdot|S_0, S_1, ..., S_t) = \pi(\cdot|S_t)$$

The policy π is said to be deterministic if all the probabilities of the resulting action distribution are concentrated on a single action or stochastic otherwise. In this thesis, we exclusively use stochastic policies modeled by Gaussian distributions. Then an action is sampled from the action distribution: $a \sim \pi(\cdot|S_t = s)$. After receiving the action a, the environment transits to the next state by sampling the transition dynamics: $s' \sim \mathcal{P}(.|S_t = s, A_t = a)$ and computing the immediate feedback called **reward** with the reward function $r = \mathcal{R}(S_t = s, A_t = a)$. The learning agent improves the policy based on a series of transition samples (s, a, r, s').

Some MDPs have a special set of states called **terminal states**. In this case, the sequence of the current agent-environment interaction ends when the environment reaches the terminal states, leaving a complete sequence called **episode**:

$$(S_0, A_0, R_1, S_1, A_1, R_1, ...S_{T-1}, A_{T-1}, R_T, S_T)$$

In this thesis, we only focus on **finite horizon MDPs**, or **episodic MDPs** in other words, in which the lengths of episodes are guaranteed to be finite.

In practice, an agent often interacts with the environment for a fixed amount of time, called **time limit**, before resetting itself and starting a new episode. This is often done to diversify the training experience of the agent by preventing the agent from being stuck in uninformative states for a long time (Pardo et al. 2018). In this setting, there are two kinds of terminal states—the goal state(s) and the states where termination is due to the time limit. Pardo et al. (2018) showed that it is necessary to distinguish these two kinds of terminations and handle them appropriately for correct value estimation. Their work suggests that we should continue bootstrapping at early terminations due to time limits. Thus, in this thesis, we consider goal states as the only true terminal states of a task.

The return of an episode starting from timestep t is defined as:

$$G_t := \sum_{t'=t}^{T-1} \gamma^{t'-t} R_{t'+1} \tag{3.1}$$

The capital letters in the above definition denote random variables since the reward at each timestep is random. The randomness comes from the uncertainty in the initial distribution μ, the transition distribution $\mathcal{P}$, and the policy π. Given the above return definition, the goal of RL for episodic MDPs can be formulated as maximizing the expected value of G_0:

$$\mathcal{J}_\pi := \mathbb{E}_\pi[G_0] = \mathbb{E}_\pi[\sum_{t=0}^{T-1} \gamma^t R_{t+1}] \tag{3.2}$$

Note that the expectation is taken over the policy π, which is the only randomness that an agent can control.

Many RL algorithms use the state-value function or the action-value function to implicitly determine policies. For example, the policy of Q-learning chooses the action with the largest action value with the probability $1-\epsilon$ and chooses a random action with the probability ϵ. The state-value function is defined as:

$$v_\pi(s) := \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{t'=t}^{T-1} \gamma^{t'-t} R_{t'+1}|S_t = s] \tag{3.3}$$

and the action-value function is defined as:

$$q_\pi(s,a) := \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{t'=t}^{T-1} \gamma^{t'-t} R_{t'+1}|S_t = s, A_t = a] \tag{3.4}$$

The superscript t indicates that the timestep t is included in the definitions in order for the Markov property to hold. Alternatively, they can also be recursively defined as:

$$v_\pi(s) := \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \tag{3.5}$$

and

$$q_\pi(s,a) := \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \tag{3.6}$$

with the assumption that $v(S_T) = 0$ and $q(S_T, \cdot) = 0$.

A group of RL algorithms called **Policy Gradient Methods** explicitly model a parameterized policy and try to maximize the objective 3.2 directly. According to the **Policy Gradient Theorem** (Sutton et al., 1999), the objective 3.2 can be converted to the following objective:

$$\mathcal{J}_\theta := \mathbb{E}_\pi\left[(\sum_{t=0}^{T-1} \log \pi_\theta(A_t|S_t))(\sum_{t'=t}^{T-1} \gamma^{t'-t} R_{t'+1})\right] \tag{3.7}$$

If the expected return $\mathbb{E}_\pi[\sum_{t'=t}^{T-1} \gamma^{t'-t} R_{t'+1}]$ from time t is estimated by Monte Carlo methods, no value function is required for both policy parameter learning and action selection. A typical example is the **REINFORCE** algorithm (Williams, 1992). On the other hand, a value function can still be used to estimate the return from time t for policy parameter learning. Methods that learn value functions to estimate returns during policy training are collectively called **Actor-Critic** methods. The two algorithms, Soft Actor-Critic (SAC) and Proximal Policy Optimization (PPO), used in this thesis are examples of Actor-Critic methods.

3.2 Soft Actor-Critic

Inspired by the maximum entropy RL, Haarnoja et al. (2018) proposed a new model-free off-policy policy gradient algorithm called the Soft Actor-Critic algorithm (SAC). Instead of maximizing the standard policy gradient objective 3.7, SAC aims to balance the maximization of the expected return and the expected policy entropy. The inclusion of policy entropy entails the use of stochastic policies. Stochastic policies have many advantages, including improved exploration (Haarnoja et al., 2017) and more stabilized training (Haarnoja et al., 2018).

As mentioned above, SAC is an actor-critic algorithm where the critic learns a parameterized entropy-augmented action-value (Q-value) function $q_\theta(S_t, A_t)$, and the actor learns a parameterized stochastic policy $\pi_\phi(A_t|S_t)$. Here θ and ϕ represent the learnable parameters of models. In this thesis, the Q-value function and policy are approximated by neural networks. The entropy-augmented action-value (Q-value) function, or the Soft Q function for short, is defined as:

$$q_\pi(s, a) := \mathbb{E}\left[r(S_t, A_t) + \gamma v(S_{t+1}) | S_t = s, A_t = a\right] \tag{3.8}$$

where $v_\pi(s) := \mathbb{E}\left[q_\pi(S_t, A_t) - \alpha \log \pi(A_t|S_t) | S_t = s\right]$.

The Soft Q function is optimized by minimizing the following mean-square error:

$$\mathcal{J}_q(\theta) := \mathbb{E}\left[(q_\theta(S_t, A_t) - (r(S_t, A_t) + \gamma \mathbb{E}[v_{\tilde{\theta}}(S_{t+1})]))^2\right] \tag{3.9}$$

In practice, training samples are sampled from the replay buffer and the target Q value should be estimated with another target network whose parameters are represented by $\tilde{\theta}$. Using a separate target network significantly increases the stability of training (Mnih et al., 2015). In addition, SAC also uses the double Q-learning trick (Hasselt et al., 2010) to reduce the positive bias.

The policy is optimized by minimizing the expected KL divergence between the new policy and the exponential of the Q function:

$$\mathcal{J}_\pi(\phi) = \mathbb{E}\left[D_{KL}\left(\pi_\phi(\cdot|S_t)||\frac{\exp(q_\theta(S_t,\cdot))}{Z_\theta(S_t)}\right)\right] \tag{3.10}$$

It can be further simplified into the following entropy-augmented objective:

$$\mathcal{J}_\pi(\phi) := \mathbb{E}[\alpha \log \pi_\phi(A_t|S_t) - q_\theta(S_t, A_t)] \tag{3.11}$$

Here α weights the importance of the entropy, and it is called the **temperature** parameter. The objective 3.11 can be optimized with the Policy Gradient Theorem. However, since our actions are continuous, it is convenient to use the reparameterization trick instead to reduce the estimation variance. SAC with reparameterization trick defines the policy as:

$$A_t = f_\phi(\epsilon_t; S_t) \tag{3.12}$$

where ϵ_t is a sampled noise from a certain distribution $\mathcal{N}$. Given the above policy representation, we can transform the objective 3.11 into:

$$\mathcal{J}_\pi(\phi) := \mathbb{E}\left[\alpha \log \pi_\phi(f_\phi(\epsilon_t; S_t)) - q_\theta(S_t, A_t)\right] \tag{3.13}$$

To keep the policy optimization tractable, possible policies are limited to a policy set $\prod$. In this thesis, policies are limited to the multi-dimensional Gaussian distribution. The complete algorithm is given in Algorithm 1 (Haarnoja et al., 2018).

3.3 Proximal Policy Optimization

The Proximal Policy Optimization algorithm (PPO) is a model-free on-policy policy gradient method proposed by (Schulman et al., 2017). It is based on the Trust Region Optimization (TRPO) method (Schulman et al., 2015) that aims to maximize a "surrogate" objective subject to a KL divergence constraint:

$$\max_\theta \mathbb{E}_t\left[\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)} H_t\right] \tag{3.14}$$

$$\text{subject to } \mathbb{E}_t\left[\text{KL}[\pi_{\theta_{old}}(\cdot|S_t), \pi_\theta(\cdot|S_t)]\right] \leq \delta \tag{3.15}$$

Algorithm 1 Soft Actor-Critic

```
Input: $\theta_1, \theta_2, \phi$                                  ▷ Initial parameters
  $\bar{\theta_1} \leftarrow \theta_1$, $\bar{\theta_2} \leftarrow \theta_2$                    ▷ Initialize target networks
  for each iteration do
      for each environment step do                              ▷ Collect samples
          $a \sim \pi_\phi(\cdot|S_t)$
          $s' \sim p_\phi(S_{t+1}|S_t, A_t)$
      end for
      for each gradient step do                ▷ Update Q network and policy
          $\theta_i \leftarrow \theta_i - \lambda_q \hat{\nabla}_{\theta_i} J_q(\theta_i)$ for i $\in \{1, 2\}$
          $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
          $\bar{\theta_i} \leftarrow \tau\theta_i + (1-\tau)\bar{\theta_i}$ for i $\in \{1, 2\}$
      end for
  end for
Output: $\theta_1, \theta_2, \phi$
```

Here θ_{old} denotes the old policy parameters, and θ denotes the current policy parameters. H_t is the advantage estimation of the current action, which is computed as:

$$\delta_t = r_t + \gamma v(S_{t+1}) - v(S_t) \tag{3.16}$$

$$H_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{3.17}$$

Nevertheless, TRPO is computationally expensive since it uses conjugate gradient ascent and a line search to enforce the hard KL divergence constraint, which makes it hard to use in the real-time learning setting.

PPO simplifies objective 3.14 to reduce the computation cost. It has two variants. The first variant treats the KL divergence as a penalty, modifying objective 3.14 into the following unconstrained objective:

$$\max_\theta \mathbb{E}_t \left[\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)} H_t - \beta \mathrm{KL}[\pi_{\theta_{old}}(\cdot|S_t), \pi_\theta(\cdot|S_t)] \right] \tag{3.18}$$

This approach is called **Adaptive KL Penalty Coefficient**. The second approach does not explicitly include the KL divergence penalty in the objective but instead implicitly respects the KL divergence by removing samples whose action probabilities are significantly different from the current policy. Specifically, this approach maximizes a **Clipped Surrogate Objective** defined as:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min(r_t(\theta)H_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)H_t)\right] \tag{3.19}$$

Here the probability ratio $r_t(\theta) = \frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)}$ for state S_t and action A_t. The clipped surrogate loss function $L^{clip}(\theta)$ clips the probability ratio to be no more than $1+\epsilon$ if the advantage is positive and no less than $1-\epsilon$ if the advantage is negative. The policy is optimized by running several iterations of stochastic gradient ascent at each update. In this thesis, we only use the second approach.

PPO is also an actor-critic method since it trains a value network to estimate the advantage of the current action A_t. The complete algorithm is given in Algorithm 2 (Schulman et al., 2017)

Algorithm 2 Proximal Policy Optimization

```
for each iteration do
    for each environment step do                                  ▷ Collect samples
        Run policy $\pi_{old}$ to collect samples
        Compute advantage estimates $H_1, H_2, H_3 \cdots, H_T$
    end for
    Optimize the clipped surrogate objective $L$ with respect to $\theta$ for $K$ epochs
end for
Output: $\theta_1, \theta_2, \phi$
```

3.4 Neural Networks

In this thesis, we employ two types of neural networks: Fully Connected Neural Networks and Convolutional Neural Networks. The subfield of reinforcement learning that exploits the capability of neural networks to map high-dimensional states such as images to actions and values is referred to as **Deep Reinforcement Learning**.

A neural network is a series of connected layers. The first layer is called the **input layer**, and the last layer is called the **output layer**. All layers in between are collectively called the **hidden layers**. Each layer consists of multiple nodes called *neurons*. If each neuron in the previous layer is connected to each neuron in the next layer, we call this network a fully connected neural network. Fig. 3.2 shows an example of a fully connected neural network.

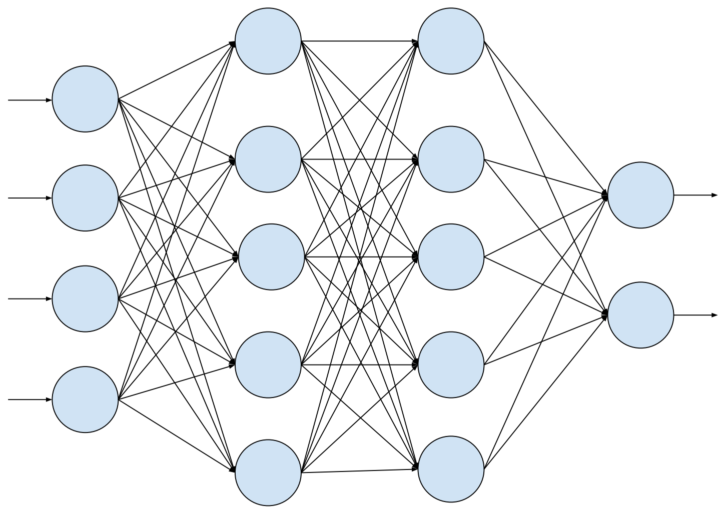


Figure 3.2: A fully connected neural network. It has two hidden layers with five neurons each, one input layer with four neurons, and an output layer with three neurons. Each arrow corresponds to a trainable parameter.

A neuron computes its output by summing up the outputs of connected neurons from the previous layer, weighted by the corresponding weights. Thus, a fully connected neural network can be mathematically modeled as a series of matrix multiplications where each layer is modeled as a matrix whose elements are weights. However, this architecture does not allow non-linear mapping as the matrix multiplication is a linear operator. To solve this problem, we often append a non-linear function called **activation** to the outputs of neurons. Some common non-linear activations used in deep learning are the Sigmoid function, Tanh function, and ReLU function. Theoretically, a fully connected neural network with non-linear activations can approximate any continuous functions to arbitrary precision (Cybenko et al., 1989; Leshno et al., 1993; Pinkus et al., 1999).

However, fully connected neural networks are often inefficient for structured data like images as it needs to be more scalable. Inspired by the structure of the animal visual cortex, Lecun et al. (1999) proposed a scalable network architecture called Convolutional Neural Networks (CNNs) to handle image inputs. CNNs exploit image data structure to reduce the number of weights and improve performance in computer vision tasks such as image classification and object localization. Besides the reduced number of weights, the main advantage of CNNs is the automatic extraction of features from images without human effort. Since CNNs share weights for different pixels, they can easily discover translational invariant features in images, which is important in computer vision tasks.

Each layer of a CNN consists of multiple kernels that transform the input volume into an output volume. A kernel is an image-alike structure containing shared weights. It performs the convolution operation on the input volume along the height and width dimensions to produce a so-called **activation map** containing the extracted features. Activation maps produced by different kernels are stacked along the depth dimension to form the output volume that will be used as the input volume for the next convolutional layer. By stacking multiple convolutional layers, the network can discover important hierarchical features for the task.

CNNs often use successive pooling operations to down-sample output volumes to discard positional information since it is usually irrelevant for computer vision tasks (Levin et al., 2016). However, positional information is important for the control policy to solve goal-reaching tasks. Thus, we do not use pooling in our image processing module.

3.5 Spatial Softmax Layer

The Spatial Softmax Layer is proposed by Levin et al. (2016) to extract positional information from the last activation map produced by a CNN, which is necessary for successfully solving target-reaching tasks. It also filters out erroneous low activations due to noise so that the training is more robust to background distractors.

The spatial softmax layer consists of two components: a spatial softmax function and an expected position function. The spatial softmax function: $s_{cij} = \frac{e_{a_{c}ij}}{\sum_{i'j'} e_{a_{c}i'j'}}$ converts an activation map to a probability distribution over locations of image features. Here, a_{cij} is the activation at the location (i, j) of the channel c, and s_{cij} is the feature probability at the location (i, j) of the channel c. Then the expected position function converts the probability distribution to the expected image positions of features by: $(f_{cx}, f_{cy}) = (\sum_{ij} s_{cij}x_{ij}, \sum_{ij} s_{cij}y_{ij})$, where x_{ij} and y_{ij} are the coordinates of the pixel (i, j) in the image space. The feature positions (f_{cx}, f_{cy}) from different channels form a vector that captures all the information retrieved from images. Later, this vector, along with additional robots' proprioception, is used as the input to the following fully connected neural networks.

3.6 Reinforcement Learning with Augmented Data

Kostrikov et al. (2020) shows that image-based RL may significantly suffer from overfitting due to a large number of parameters of the image encoders. Hence we adopted the technique called Reinforcement Learning with Augmented Data (RAD) proposed by Laskin et al. (2020) to mitigate the overfitting problem. Based on their results, we use the **Crop** augmentation method that performs the best in most tests.

The Crop augmentation method randomly extracts a smaller image patch from the original image and feeds the extracted image patch to neural networks as the image observation. For example, if the original image size is $H \times W \times C$ (*height, width, channel*), the Crop augmentation randomly crops it into an image of the size $H' \times W' \times C$. In this case, the starting point (cy, cx) for cropping is sampled from the uniform distribution $(\, U(0, H - H'), U(0, W - W'))$. For the convenience of discussion, we define the height cropping margin $h_y = \frac{H-H'}{2}$ and the width cropping margin $h_x = \frac{W-W'}{2}$). In this work, we use cropping margin $h_y = 0.1 \times H$ and $h_x = 0.1 \times W$. Fig. 3.3 demonstrates the cropping process.

A caveat is that image augmentation is only applied to training samples to increase sample diversity and reduce overfitting during learning. We always use the central image patch to compute actions for the agent-environment interaction, i.e., the starting point for cropping is always (h_y, h_x). For sim-

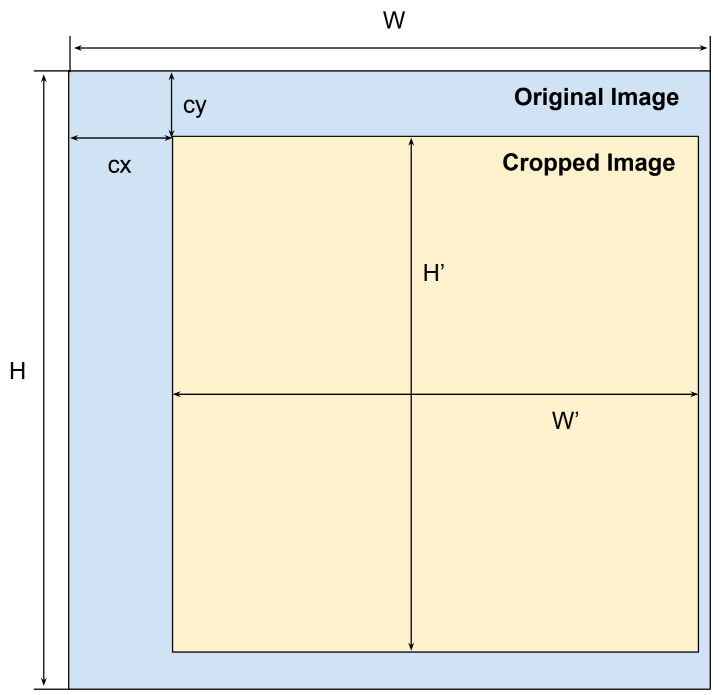


Figure 3.3: The cropping augmentation

plicity, cropping is consistent within a mini-batch and random across different mini-batches.

3.7 SenseAct

SenseAct is an implementation of the computational framework for designing real robotic tasks proposed by Rupam et al. (2018a). Unlike discrete simulation timesteps, the time of the real-world marches during action computation and policy update, which imposes additional challenges to real-time learning since the agent will have to work with delayed observations. The SenseAct framework mitigates this problem by carefully ordering and parallelizing computations of agent-environment interactions.

Since Python does not support true multi-threading parallelism, concurrent computations are implemented with processes.

Fig. 3.4 shows the high-level architecture of SenseAct. Each robotic task has at least three concurrent processes. Please note that a robotic task can have more than one communicator process for complex control.

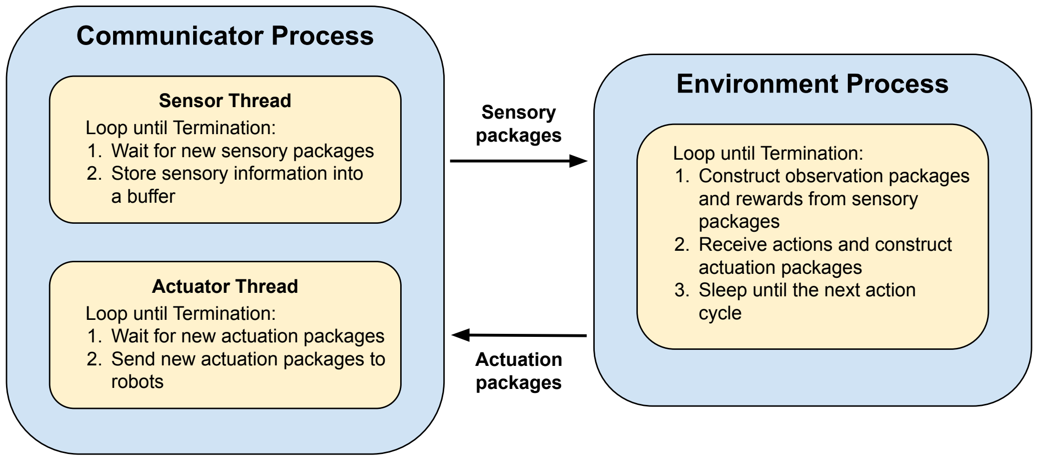

Communicator Process
Sensor Thread
Loop until Termination:
1. Wait for new sensory packages
2. Store sensory information into a buffer
Actuator Thread
Loop until Termination:
1. Wait for new actuation packages
2. Send new actuation packages to robots
Sensory packages
Actuation packages
Environment Process
Loop until Termination:
1. Construct observation packages and rewards from sensory packages
2. Receive actions and construct actuation packages
3. Sleep until the next action cycle

Figure 3.4: The architecture of SenseAct

Chapter 4

Distribution of Computations

In this chapter, we investigate the first issue of real-time learning of vision-based robotic tasks in the real world: How much the performance of a learning system will be affected by limited by resources limitations and how to distribute computations of a learning system to best compensate for performance loss due to limited onboard resources. **Our hypothesis is two-fold. First, the performance of resource-demanding learning systems will degrade substantially on the resource-limited computer compared to that of the powerful workstation. Second, carefully chosen distributions of computations will substantially compensate for performance loss, especially for resource-demanding learning systems.** To test our hypotheses, we proposed the Remote-Local Distributed (ReLoD) system and developed two visuomotor robotic tasks.

4.1 ReLoD System

Most deep RL systems have three computationally expensive components: 1) Action Computation, 2) Policy Updating, and 3) Minibatch Sampling. ReLoD spawns parallel processes for each of the three components and runs them in a distributed manner to better utilize the computational resources of the local and remote computers. In addition, it creates additional inter-process communication sockets to enable this distributed and parallel architecture (described later). ReLoD supports three different modes of operation: *Remote-Only*, *Remote-Local*, and *Local-Only*. The mode of operation determines how the various processes are distributed between the local computer and the remote computer.

ReLoD is designed to work with various RL algorithms. Users can easily add more RL algorithms to the system by implementing the agent interface. In this thesis, we implement two state-of-the-art RL algorithms: SAC and PPO. We call the resulting agents *SAC-agent* and *PPO-agent*, respectively.

4.1.1 Description of Remote-Local SAC-agent

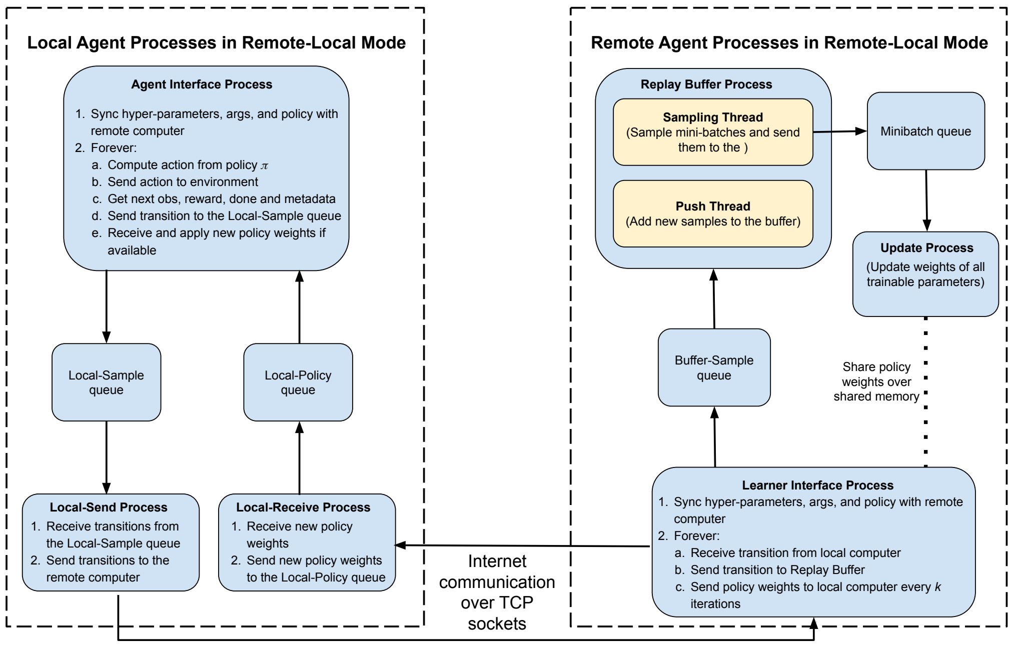

Local Agent Processes in Remote-Local Mode
Agent Interface Process
1. Sync hyper-parameters, args, and policy with remote computer
2. Forever:
a. Compute action from policy π
b. Send action to environment
c. Get next obs, reward, done and metadata
d. Send transition to the Local-Sample queue
e. Receive and apply new policy weights if available
Local-Sample queue
Local-Policy queue
Local-Send Process
1. Receive transitions from the Local-Sample queue
2. Send transitions to the remote computer
Local-Receive Process
1. Receive new policy weights
2. Send new policy weights to the Local-Policy queue
Internet communication over TCP sockets
Remote Agent Processes in Remote-Local Mode
Replay Buffer Process
Sampling Thread
(Sample mini-batches and send them to the)
Push Thread
(Add new samples to the buffer)
Minibatch queue
Update Process
(Update weights of all trainable parameters)
Buffer-Sample queue
Share policy weights over shared memory
Learner Interface Process
1. Sync hyper-parameters, args, and policy with remote computer
2. Forever:
a. Receive transition from local computer
b. Send transition to Replay Buffer
c. Send policy weights to local computer every k iterations

Figure 4.1: The architecture of the Remote-Local mode of SAC-Agnet

Fig. 4.1 outlines the structure of the SAC-agent in the Remote-Local mode. There are three processes, called *Agent-Environment Interface*, *Local-Send*, and *Local-Receive*, that run in parallel on the local computer. The Agent-Environment Interface process computes the actions to take in the environment (except in Remote-Only mode given in Table 4.1). The Local-Send process sends the agent-environment interaction sample $(S_t, A_t, R_{t+1}, S_{t+1})$ at timestep t to the remote computer via a TCP connection.

Three more processes, called *Learner Interface*, *Replay Buffer*, and *Update*, run in parallel on the remote computer. The Replay Buffer process samples mini-batches, and the Update process updates policies. Considering that neither the asynchronous SAC described in 4.2 nor PPO requires heavy operations in the Learner Interface process, we do not use a separate process to receive samples from the local computer for simplicity since such a process will not significantly increase the reception rate. Please remember that the term *asynchronous* in this thesis and asynchronous SAC means using concurrency to parallelize the computations of a learning system. By distributing the three most computationally expensive components of SAC on two different computers, ReLoD takes advantage of both computers. On the other hand, in the Remote-Only mode, all computations happen on the remote computer, including action computation. The local computer, in this case, only relays observations and actions between the remote computer and the robot. More details about the process distribution for all three modes of the SAC-agent are given in Table 4.1.

Note that the learning algorithm determines what processes to use in each mode. For example, the Replay Buffer and the Update processes are specific to our asynchronous SAC implementation adapted from the system proposed by Yuan and Mahmood (2022). The PPO-agent does not have them due to the nature of the algorithm. It updates policies in the Learner Interface process instead. Please note that in the Remote-Local mode, allocating expensive operations on the resource-limited computer is impractical. Since the local computer is usually the resource-limited computer and policy updating is way more expensive than action computation, the only meaningful distribution in the remote-local mode is deploying policy updating on the remote computer and deploying action computation on the local computer.

The Remote-Local mode is especially suitable for real-world applications where sufficient onsite computational power is hard to achieve. A typical use case is deploying robotic arms in a rural warehouse to pick up items automatically. Although the Remote-Local mode has been widely used in the industry, no systematic study in the literature shows that this mode will significantly improve performance.

	Remote-Local	**Remote-Only**	**Local-Only**	**Period of Iteration**
Agent Interface Process (AIP)	Local	Local	Local	action cycle time
Action computation	Included in AIP	Included in LIP	Included in AIP	action cycle time
Local-Send Process	Local	N/A	N/A	action cycle time
Local-Receive Process	Local	N/A	N/A	Every k iterations of LIP's loop
Learner Interface Process (LIP)	Remote	Remote	Included in AIP	action cycle time
Replay Buffer Process	Remote	Remote	Local	hardware dependent
Update Process	Remote	Remote	Local	hardware dependent

Table 4.1: A list of all processes and their distributions between the local and the remote computers in various operational modes of the ReLoD system.

4.1.2 Inter-Process and Internet Communication

Due to Python limitations, ReLoD is designed with multi-process parallelism, which imposes additional technique difficulties on efficient Inter-Process communications (IPCs). For simplicity, we use the shared queue for IPCs exclusively within the same computer. As Fig. 4.1 shows, ReLoD creates four shared queues for the Remote-Local SAC-agent. The Local-Sample queue on the local computer and the Buffer-Sample queue on the remote computer are created with *length = time limit* to prevent losing samples and blocking the agent-environment interaction. The Local-Policy queue on the local computer and the Minibatch queue on the remote computer are configured to the non-blocking mode since policy update is asynchronous. Their lengths can be any value larger than 1. In this work, we arbitrarily set their lengths to 2.

ReLoD uses TCP sockets for Internet communications. The main benefit of using TCP sockets is that the TCP protocol guarantees packets' delivery and the order of arrival, allowing a more efficient way of transmitting samples. Since S_t is just a duplicate of the previous sample, if the order of arrival of packets can be guaranteed, the local computer only needs to send compact samples (A_t, R_{t+1}, S_{t+1}) at every timestep to reduce the bandwidth requirement by approximately half compared to transmitting complete samples $(S_t, A_t, R_{t+1}, S_{t+1})$. The initial state s_0 of a new episode is sent before the new episode starts. This is a very appealing feature in practice because images take more than 90% of the total observation data. However, the drawback is that ReLoD will only work efficiently with good networking quality due to the nature of TCP connections.

It is also possible to use UDP sockets instead of TCP sockets at the cost of increased bandwidth. That is because the UDP protocol does not guarantee packets' delivery and the order of arrival. As a result, we will have to send both the previous and the next states, even if they are duplicates. Considering that terminal states are critical for learning correct value functions and may be sparse in practice, it is better to use TCP sockets to transmit samples involving terminal states to avoid losing them.

4.2 The Learning System

In this thesis, ReLoD system was used to distribute two state-of-the-art real-time reinforcement learning systems: 1) the asynchronous SAC proposed by Yuan and Mahmood (2022), and 2) the spinup PPO (Achiam, 2018). The two learning systems were extended to support our ReLoD system. In addition, we modified the asynchronous SAC implementation to allow bootstrapping on the time limit states when updating the value function. For example, if an interaction sequence $(S_0, S_1, \cdots, S_t)$ terminates prematurely at the state S_t due to time limits, the original implementation updates the value of the state S_{t-1} with $v(S_{t-1}) = R_t$. That is, the value of the state S_t is assumed to be 0. Instead, our modified implementation updates the value of the state S_{t-1} with: $v(S_{t-1}) = R_t + \gamma v(S_t)$. That is, the estimated value of the state S_t is used to update S_{t-1}. The spinup PPO already supports this update in the original implementation. This modification is necessary since we define an episode to be a sequence from an initial state S_0 to a terminal(target) state S_T. In this setting, the time limit states are not the real terminal(target) states, and thus their state values should not be assumed to be 0.

We used the same neural network architecture proposed by Yuan and Mahmood (2022) to parameterize policies. It includes three networks: **Image Encoder Network**, **Actor Network**, and **Critic Network**. The image encoder network is only used for image-based tasks. For tasks that do not use images, we remove the image encoder to save resources. The interaction of the three networks is shown in Fig. 4.2. The design details of the three networks

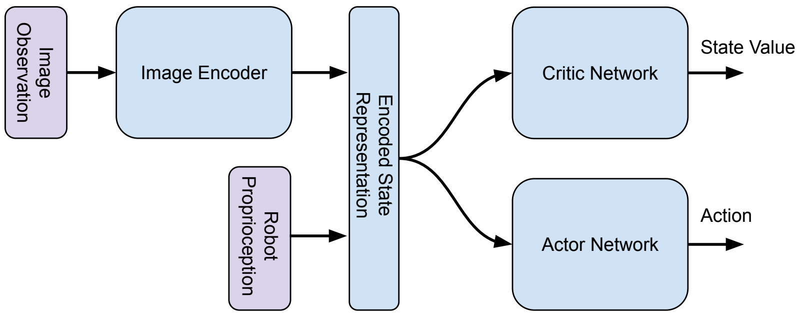


Figure 4.2: The structure of the neural networks

are discussed in the following subsections. The weights of neural networks are adjusted by the learning systems to improve policies. Their relationship is shown in Fig. 4.3.

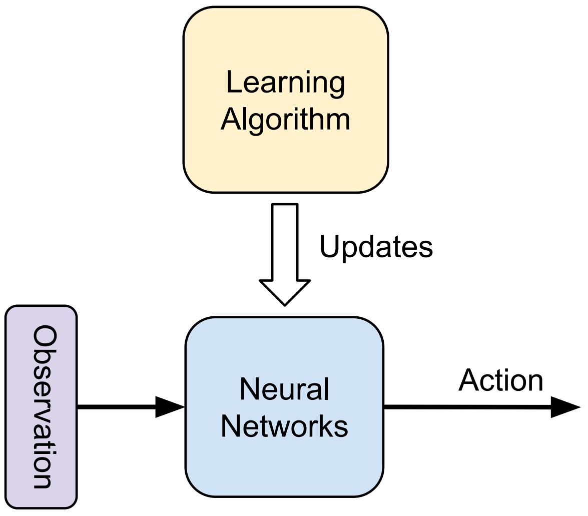


Figure 4.3: The interaction between neural networks and the learning system

4.2.1 Image Encoder Network

The image encoder network extracts information from images. It consists of two components: a CNN and a spatial softmax layer. The CNN has four layers, each containing 32 kernels of the size 3×3. The input layer and the two hidden layers use $stride = 2$, and the output layer uses $stride = 1$. During the training phase, images are randomly cropped according to the RAD technique described in section 3.6. The activation map of the last layer is then fed into the spatial softmax layer.

The spatial softmax layer computes the expected feature positions (f_x, f_y) for each channel of the final activation map, resulting in 32 pairs of (f_x, f_y). Finally, the 32 pairs of (f_x, f_y) are flattened into a vector of size 64 and concatenated with the robot's state vector for the actor and critic Networks.

4.2.2 Actor and Critic Networks

In this work, the actor and the critic networks share the same architecture except for the input and output layers for simplicity, though they can have different architectures in general. Specifically, they have two fully connected hidden layers with 1024 neurons each.

Obviously, the size of the input layer will vary with tasks. The critic network has a simple output layer in which there is only one neuron since it is used to estimate either the state value or the Q value. The size of the actor network's output layer depends on the number of actions and the model of action distribution. We use the multivariate normal distribution in this work to model the action distribution:

$$\frac{\exp(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}))}{\sqrt{(2\pi)^k|\boldsymbol{\Sigma}|}} \tag{4.1}$$

Since we assume that action components are independent of each other, the covariance matrix $\boldsymbol{\Sigma}$ is diagonal. In this case, each action component needs two parameters: the mean μ and the variance σ. Therefore, the output layer size of the actor network is $2 \times k$, where k is the action dimension. In this work, we initialize the weights of the last layer of the actor network to zeros so that the initial policy is the Gaussian distribution $\mathcal{N}(0, 1)$ for all tasks.

4.3 Tasks

We developed two visuomotor robotic tasks, named *UR5-VisualReacher* and *Create-Reacher*, to test our hypotheses in this chapter. The two tasks are not easy to solve using simple RL approaches since they require tens of thousands of images to learn good state representations and control policies. The robots we used are the **UR5** industrial robot arm and iRobot **Create2** mobile robot. Please note that, the two robotic tasks require SenseAct (Rupam et al., 2018a) to run. They are designed using the guidelines proposed by Mahmood et al. (2018a) to reduce the computational overhead and take advantage of the multi-processing capability of modern computers for handling different transmission rates. Task details are given below.

4.3.1 UR5-VisaulReacher

UR5 is an industrial robotic arm manufactured by Universal Robotics. It has six joints, and each joint can be controlled individually. We do not actuate the end-effector joint as there is no gripper attached. UR5 uses conventional TCP/IP protocol to communicate packets with the host computer every 8ms. A state packet contains the six controllable joints' angles, velocities, target accelerations, and currents. UR5 can be configured to use either the velocity control or the angular control modes. Users can send low-level control commands directly to the six joints in both modes, making UR5 an ideal platform for reinforcement learning studies. In this thesis, we used the velocity control mode.

Since a UR5 robotic arm does not come with a camera, we attached a Logitech RGB camera to the tip of the arm to enable image-based tasks. This task is the same one proposed by Yuan and Mahmood (2022). Fig. 4.4 shows the task setup. It is an image-based task that aims to move a UR5 arm's fingertip to a random target designated by a red (the target color) blob on a monitor. The movement of the UR5 arm is restricted within a bounding box to prevent collisions and accidental damage. The action space is the desired angular velocities for the five joints between [-0.7, 0.7]rad/s. The observation vector includes joint angles, joint angular velocities, three consecutive images of dimension $160 \times 90 \times 3$ taken by a camera, and the previous action. Every 40ms, the SenseAct environment process actuates the five joints and receives the next observation by sending low-level control commands. The SenseAct communicator process transmits joint data every 8ms and image data every 33.¯3ms to the environment process. Since this environment is inherently non-

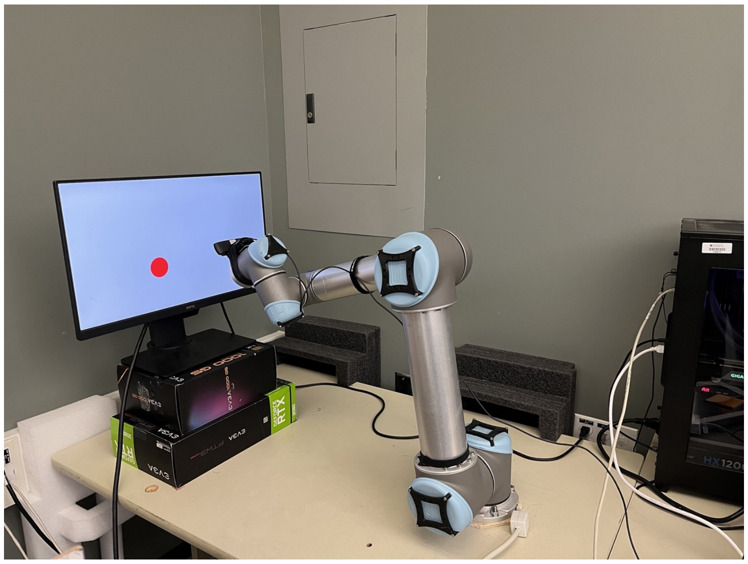


Figure 4.4: UR5 VisualReacher

Markovian, we stack three images to provide the partial history to the learning agent. Each episode lasts 4 seconds (i.e., 100 timesteps). The reward function is designed to encourage the following behaviors: 1) moving the fingertip closer to the target while keeping the target centralized, and 2) avoiding abrupt twisting of the joints. Specifically, the reward function is defined as follows (Yuan & Mahmood, 2022)

$$r_t = \alpha \frac{1}{hw} M \circ W - \beta \left(\left| \pi - \sum_{n=1}^{3} \omega_n \right| + \left| \sum_{n=4}^{5} \omega_n \right| \right),$$

where h is the height and w is the width of the image observation, M is a mask matrix whose element $M_{i,j}$ is 1 if the target color is detected at location (i, j) and 0 otherwise, W is a constant weight matrix whose elements range from 0 to 1 to encourage centralization of the target, and ω is the vector of the five joints' angles. The coefficient α and β weigh the importance of the two behaviors. In this paper, we set $\alpha = 800$ and $\beta = 1$. During a reset, a new random target will be displayed on the monitor for the next episode, and the UR5 arm will be set to a predefined posture. The environment runs on a Lenovo Ideapad Y700 laptop.

4.3.2 Create-Reacher

Create2 is a robotic development platform based on a commercially available iRobot Roomba vacuum cleaner. Unlike Roomba, Create2 has an open interface to allow custom applications, including reinforcement learning studies. Create2 has two controllable wheels and various onboard sensors. It has six wall sensors to measure the distances to walls, three infrared sensors to locate its charger, four cliff sensors to detect falling edges, two bump sensors to detect bumping into obstacles, and one charging sensor to detect successful docks. The iRobot Create2 Open Interface streams its sensory data and internal states to the host computer every 15ms. Users can send the desired speeds of the two wheels to Create2 via the open interface. Like UR5, we attached a Depstech 4K camera and a Jetson Nano 4GB computer to it to enable image-based tasks and onboard inference.

Create-Reacher is an image-based task that aims to move Create2 as soon as possible to one of the two targets designated by green papers attached to walls. The setup is shown in Fig. 4.5. The target is thought to be reached if it

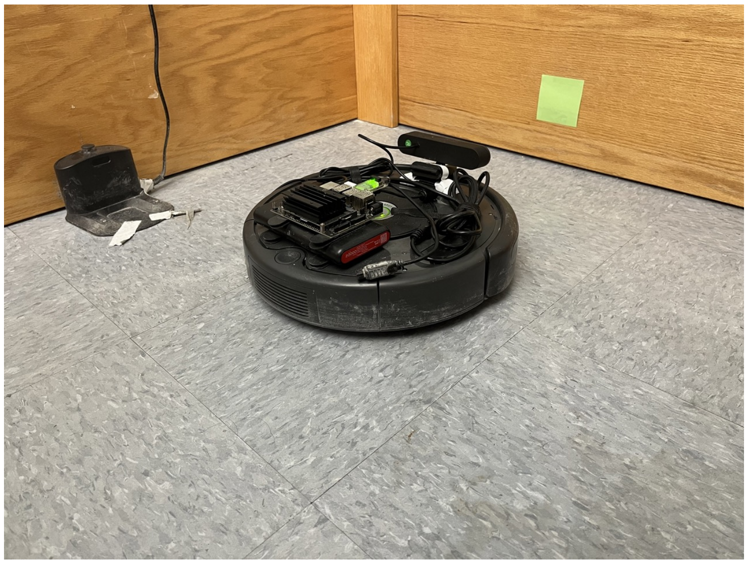


Figure 4.5: Create Reacher

occupies more than 12% pixels of the current image. The size of the image is $160 \times 120 \times 3$. The movement of Create2 is limited by an arena of size 220cm $\times$ 140cm built of wood and cardboard. The action is the desired speed for the two wheels, limited to [-300, 300]mm/s The observation vector includes the values of the six wall sensors, three consecutive images taken by a camera, and the previous action. Every 45ms, the SenseAct environment process actuates the two wheels and receives the next observation by sending low-level control commands. The SenseAct communicator process transmits the sensory data every 15ms and the image data every $33.\bar{3}$ms to the environment process. Similar to UR5-VisualReacher, we stack three images to give the partial history to the learning agent. The reward is -1 per step to encourage shorter episodes. The current episode will terminate earlier if Create2 reaches the target. If Create2 does not reach a target within the time limit, the reset routine will bring Create2 to a random location in the arena. During a reset, Create2 first moves backward and reorients itself in a random direction. Our reset routine also handles recharging when the battery life falls below a specified threshold. This environment runs on a Jetson Nano computer.

4.4 Experiment Setup

We first compared the performance of Local-Only SAC between a resource-limited local computer and a powerful local workstation, referred to as *Local-WStation-Only*, to evaluate the impact on performance due to resource limitations. We used the IdeaPad Y700 laptop as the resource-limited computer. It has an Intel i7-6700HQ CPU, an NVidia GTX 960M GPU, and 16GB memory. We reduced SAC's mini-batch size to 64 and the maximum allowed buffer size to 16000 on the laptop to fit the hardware constraints. We empirically found that a buffer size larger than 16000 would not significantly improve performance.

In addition, we also aimed to investigate how SAC would have performed on a Jetson Nano embedded computer since heavily resource-limited computers like Jetson Nano are commonly used as the local controlling computers for small robots. However, when ReLoD was configured to Local-Only SAC, Jetson Nano could not run the system fully onboard as its hardware and software architectures were not optimized for updating network weights. Jetson Nano took about half a second to complete one network update on average and became unresponsive after a couple of updates. To overcome the architectural limitations and get an empirical upper bound of performance on such a heavily resource-limited computer, we emulated the capability of Jetson Nano with the IdeaPad Y700 laptop by restricting its update rate and available memory. Thus, two Local-Only variants of SAC were tested on the laptop:

- *Local-Laptop-Only*: This variant fully utilizes the computational resources of the laptop. The policy is updated at the fastest allowable speed, that is, back-to-back.

- *Local-JNanoEm-Only*: This variant emulates the capability of Jetson Nano. Since the average time required for one SAC update on Jetson Nano is $500ms$ and the action cycle time of UR5-VisualReacher is $40ms$, the policy is updated once per 12 steps on the laptop to best match the capability of Jetson Nano. The buffer size is set to 16000, a maximum that fits Nano's 4GB RAM.

We used UR5-VisualReacher for this comparison, as mounting the laptop to Create2 was impractical. The workstation for UR5-VisualReacher has an AMD Ryzen Threadripper 2950 processor, an NVidia 2080Ti GPU, and 128G memory.

Then we investigated how to utilize the resource of a remote workstation to compensate for the performance loss due to the resource-limited local computer since tethering a workstation to a mobile robot is inconvenient in practice. We used the most common setting in practice in which the robot is tethered to a small resource-limited computer and wirelessly connected to a powerful workstation. Two typical algorithms, SAC and PPO, were tested to get better coverage of different types of RL algorithms.

SAC is resource-demanding since it updates the policy at every step and requires a large replay buffer to store all its past experiences. PPO is relatively simpler since it only updates the policy every few thousand steps, and its buffer needs to hold only a few episodes generated on-policy. The SAC-agent was tested on UR5-VisualReacher and Create-Reacher, while the PPO-agent was only tested on UR5-VisualReacher. We used a Jetson Nano 4GB for Create-Reacher and the laptop for UR5-VisualReacher as the local computer. The workstation for Create-Reacher has an AMD Ryzen Threadripper 3970 CPU, an NVidia 3090 GPU, and 128G memory.

Finally, on a powerful tethered workstation, we compare the performance of Local-Only SAC to that of the system proposed by Yuan and Mahmood (2022), which is well-tuned for a single computer. This test is to determine if the performance of the Local-Only mode of our system is comparable to the off-the-shelf RL algorithm implementation given adequate resources. The testing environment is UR5-VisualReacher, and five independent runs were performed for each system.

Chapter 5

Results of Distribution of Computations Experiments

In this chapter, we show and discuss the results of the computation distribution experiments. Most importantly, our results confirm the two-fold hypothesis proposed in chapter 4 [1]

5.1 Performance of SAC on Resource-Limited Computer

The results of SAC on resource-limited computers are shown in Fig. 5.1. It shows that the performance of SAC on the resource-limited laptop dropped by about 28% on average compared to that of SAC on the workstation. Moreover, the Local-JNanoEm-Only variant struggled to learn a comparable policy within the same time frame. Note that SAC's Local-WStation-Only configuration attained the highest return among all our results. **Our results confirmed our hypothesis that the performance of resource-demanding learning systems would degrade substantially on the resource-limited computer compared to that of the powerful workstation.**

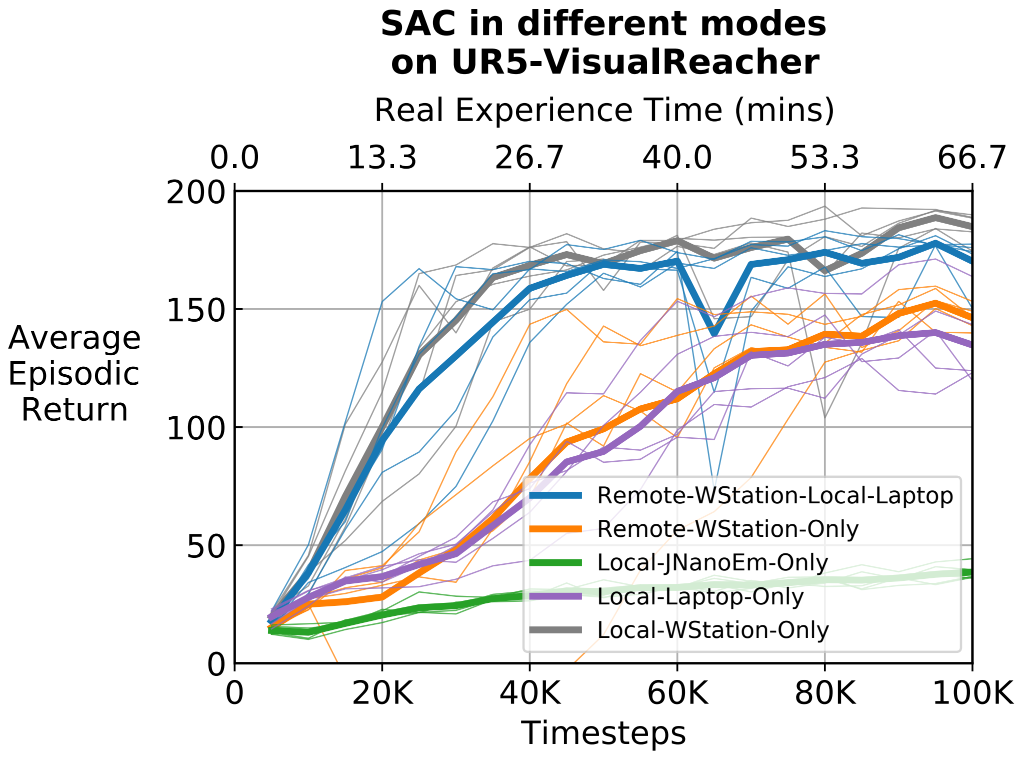


Figure 5.1: The comparison of the learning performance of the SAC-agent. The wide lines for each mode are averaged over five independent runs. The real experience time in our plots does not include any additional components required to run an experiment, such as resetting the environment or recharging the robot

5.2 Performance of SAC and PPO across Three Distribution Modes

The learning curves of the three modes of the SAC-agent on UR5-VisualReacher and Create-Reacher are shown in Fig. 5.1 and Fig. 5.2, respectively. Note that due to the hardware limitations of Jetson Nano, SAC-agent's Local-Only mode was not run for Create-Reacher. Fig 5.1 reveals a counter-intuitive result that SAC's Remote-Only mode, referred to as *Remote-WStation-Only* in the Figures, barely improved its performance over the Local-Only mode, which used a resource-limited laptop. Meanwhile, the Remote-Only mode also exhibited higher variance in the overall learning performance on both tasks. This is sur-

[1]Eexperiment videos can be found at `https://youtu.be/7iZKryi1xSY`

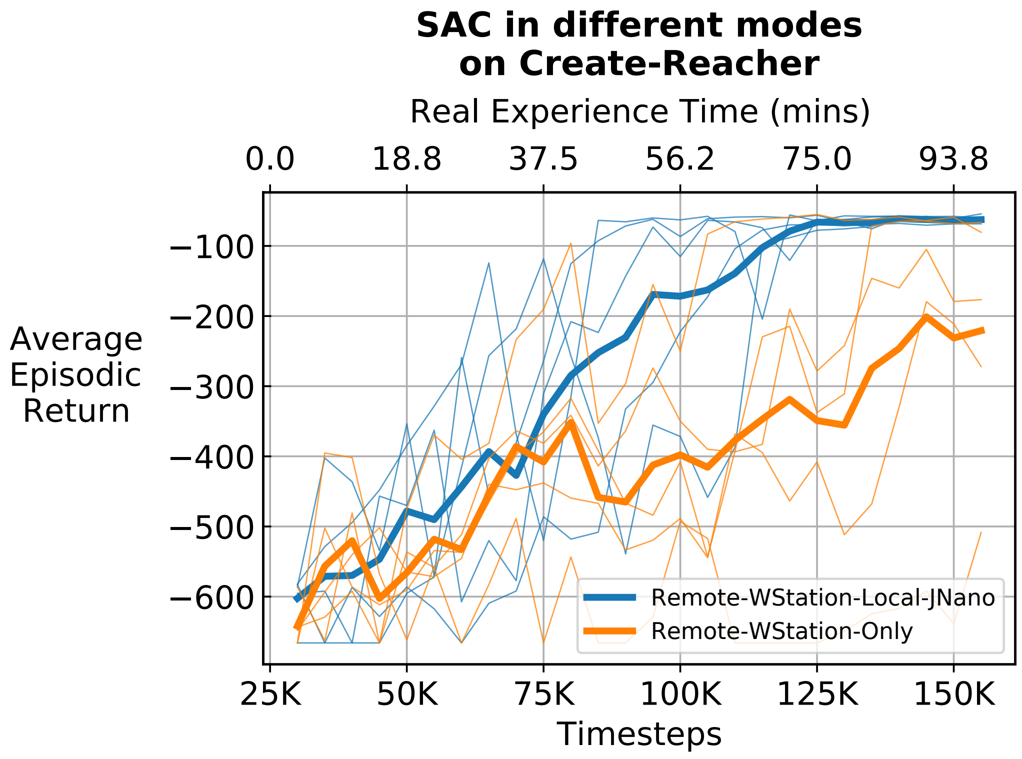


Figure 5.2: Comparison of learning performance across two modes of SAC-agent on Create-Reacher. The wide lines for each mode are averaged over five independent runs

prising since this mode fully utilizes all resources of the remote workstation. We attribute the relatively poor performance of the Remote-Only mode to the variable latency in communicating actions over WiFi, which will not occur in the other two modes, as the Remote-Only mode computes actions on a remote computer. Although the Remote-Local mode also encounters latency due to WiFi communication, the effect is much less severe as it only involves the transfer of policies and buffer samples, while action computation still happens locally. Our results align with Mahmood et al.'s (2018a) results showing that delays occurring closer to robot actuation can be substantially more detrimental to performance than delays occurring further from actuation. A real-time learning system can get affected by delays at least through two pathways: inference for action computation and learning update. When a delay occurs within the system, it may spare the more important inference pathway, as in our case. When a delay occurs closer to the robot actuation, that is, at the periphery of the learning system, it affects both pathways. Our results reveal that without careful consideration, using a powerful remote computer may not result in performance improvement.

Fig. 5.1 and Fig. 5.2 suggest that SAC-agent's Remote-Local mode, referred to as *Remote-WStation-Local-Laptop* and *Remote-WStation-Local-JNano* in the Figures, consistently compensated for the performance loss on both tasks. As Fig. 5.1 indicates, the highest average return attained by SAC-agent's Remote-Local mode is more than 90% of the highest average return attained by SAC in the Local-WStation-Only configuration, whose performance is the best. However, the Remote-Local mode only benefits computationally intensive algorithms like SAC, as Fig. 5.3 shows that PPO's performance is nearly the same across the three modes. **Our results confirmed our hypothesis that carefully chosen distributions of computations would substantially compensate for performance loss, especially for resource-demanding learning systems.**

By comparing Fig. 5.1 and Fig. 5.3, we notice that SAC significantly outperformed PPO on complex robotic tasks. All distribution modes of the SAC-agent except the Local-JNanoEm-Only attained more than twice the return attained by PPO in a shorter time frame. Nevertheless, when all computations of SAC happen on a computer incapable of performing frequent policy updates, as indicated by Local-JNanoEm-Only, SAC's performance degrades substantially to a degree comparable to that of PPO on the same computer. The poor performance of PPO can be partially attributed to the fewer policy updates. As Fig. 5.3 shows, PPO's learning curves did not plateau at the end of training. Thus, it is possible that PPO's performance would eventually match SAC. Our results indicate that SAC learns faster but needs powerful machines to be effective. On the other hand, PPO learns slower but does not require powerful computers. There is a clear trade-off between the learning speed and the availability of computational resources. In practice, the choice of learning algorithm should depend on the availability of training time and computational resources.

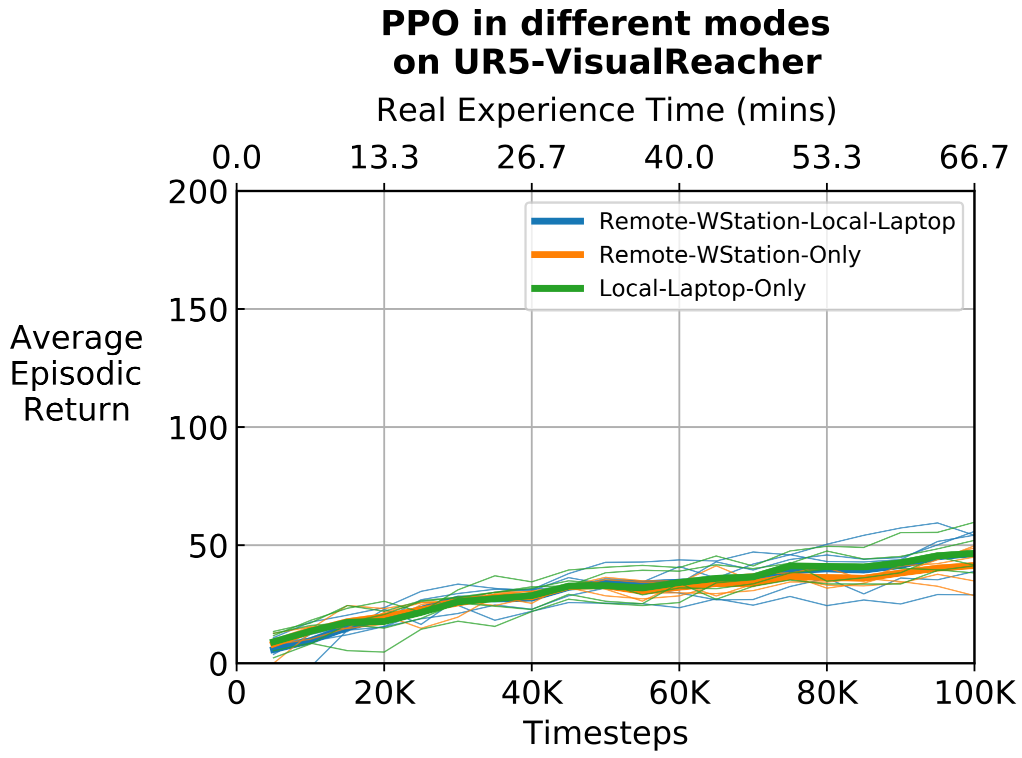


Figure 5.3: Comparison of the learning performance of the PPO-agent.

5.3 Comparison of Local-Only SAC and Off-the-Shelf System

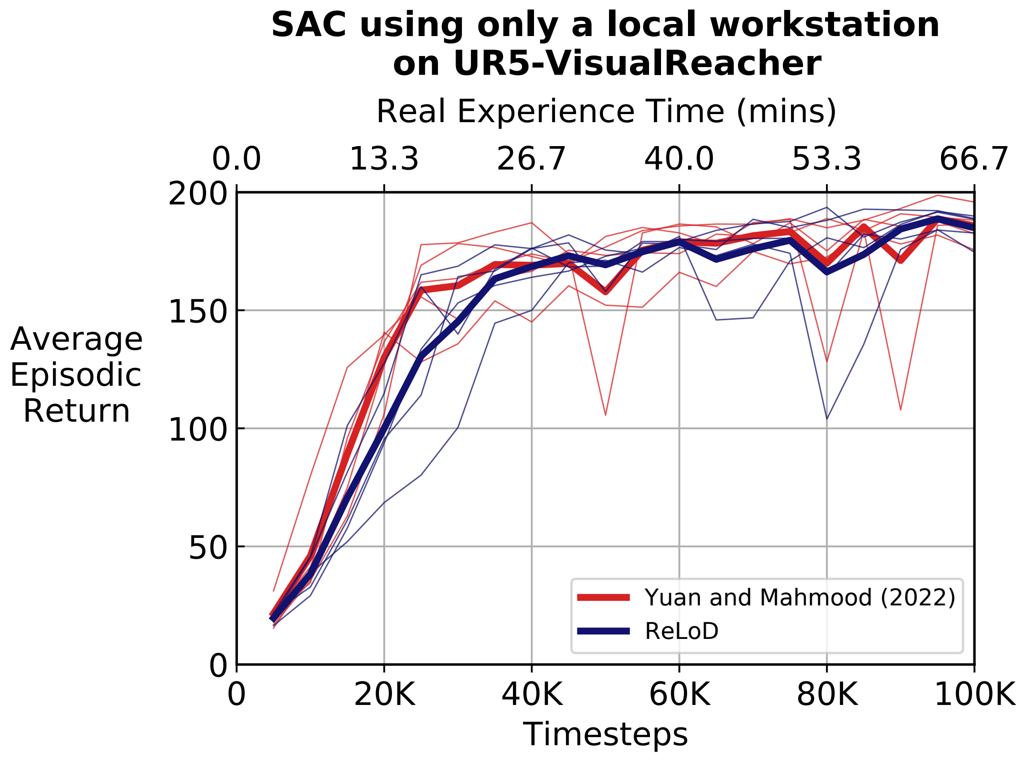


Figure 5.4: comparison of the Local-Only mode of the SAC-agent with an off-the-shelf asynchronous SAC implementation proposed by Yuan and Mahmood (2022) on a powerful workstation.

The comparison results are shown in Fig. 5.4. Both methods achieved similar overall learning performance. Thus, our proposed system can leverage the power of deep RL algorithms effectively, and as a result, it is also suitable for conventional RL research.

Chapter 6

Minimum-Time Tasks

In this chapter, we provide a primitive investigation into the second issue of real-time learning of vision-based robot tasks in the real world: How to design a reward function that is easy to define but still captures the desired behavior. It is a primitive investigation because we limit the scope to propose guidelines to determine if a minimum-time task is solvable by SAC using the performance of the initial policy before the actual training.

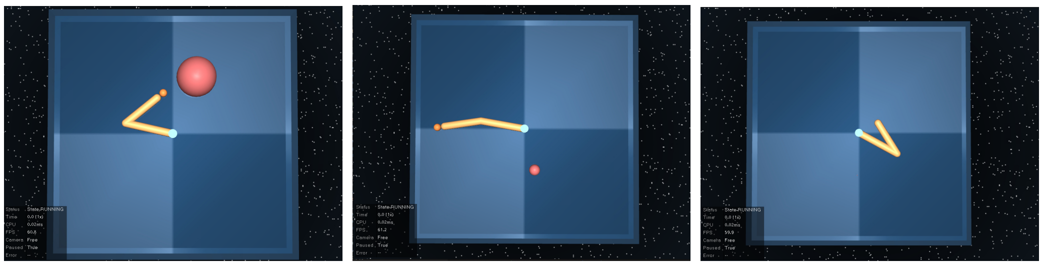
6.1 Minimum-Time Formulation

First of all, we formulate Minimum-Time tasks in this section. For all minimum-time tasks, an episode is said to be completed when the agent reaches the terminal state. More specifically, we define an episode as a set of transitions $\{(S_{t-1}, A_t, S_t)|1 \leq t \leq T'\}$, where S_0 is the initial state of the episode and $S_{T'}$ is the terminal state. If the agent does not reach the terminal state within a given time limit, we reset the agent and environment in such a way that the terminal state remains the same, but the agent moves to a different starting state. In this thesis, we used the random reset function to move the agent to an arbitrary random state. Resetting the agent to an arbitrary random state instead of predefined initial states will benefit exploration since the agent will have more chances to start from states closer to the terminal states. It is easy to show that the simple exploration strategy with a random reset function and $\mathcal{N}(0, 1)$ initial policy have a non-zero probability of finding the solution if it exists. It is possible to use other reset functions. Although smarter reset functions may improve learning performance, searching for better reset functions is not our focus. In this thesis, we focus on showing that simple reset functions, such as the random reset function with a proper time limit, can solve complex minimum-time tasks.

In this thesis, the invocation of the reset function is called *timeout*, and the maximum allowable steps before a timeout is called the *time limit*. Since resetting the agent has an associated time cost in the real world, we penalize the agent when it fails to reach the terminal state within the time limit. The penalty is the negative of the number of timesteps required to reset the agent. In addition, we include reset steps into the total steps of the current episode. Including all reset steps instead of treating reset as a single step allow us to alter the time limit without changing the problem MDP. The episodic returns and lengths are adjusted appropriately. For example, consider a task where the reward is -1 each timestep, the time limit is 100 steps, and reset penalty $=$ -20. If an agent times out thrice consecutively and finally reaches the terminal state in 25 steps since the last timeout, then the return of the episode, in this case, is $-100 - 20 - 100 - 20 - 100 - 20 - 25 = -385$, and the length of the episode is 385. Our learning curves use undiscounted returns calculated in this manner.

6.2 Learning of Minimum-Time Tasks

In this section, we show the preliminary results of using SAC to solve minimum-time tasks. We developed three simulated minimum-time tasks for this thesis. They are named *Ball-in-Cup*, *Reacher* and *Dot-Reacher*. Details about the Ball-in-Cup task and the Dot-Reacher task will be given in the next chapters. The minimum-time Reacher task was derived from the Deepmind Control Suite (Tassa et al., 2018). The original Reacher uses guiding rewards to encourage a shorter distance between the fingertip and the target. It is important to note that episodes do not end when the agent arrives at the terminal state. The agent can continue accruing rewards for the duration of the episode (i.e., a fixed time limit of 1000). We converted it into the minimum-time formulation,



(a) Reacher - Easy (b) Reacher - Hard (c) Reacher - Extreme

Figure 6.1: Three variants of the Reacher task from DeepMind Control Suite

i.e., a reward of -1 will be given to the agent each timestep until termination. Besides the necessary changes to the reward functions and termination conditions, we also modified the task formulations to reflect typical setups of real robot tasks. Minimum-Time Reacher has three sub-tasks with different target sizes representing three difficulty levels. Each sub-task has two variants: visual and non-visual tasks. Visual tasks require the agent to learn useful information from images, which are more challenging than non-visual tasks.

6.2.1 Minimum-Time Reacher Task

This task aims to move the fingertip of a planar arm with two degrees of freedom (DoF) to a random spherical target on a 2D plane. It has three sub-tasks: easy, hard, and extreme. They differ in the sizes of the target and the fingertip of the arm (see Fig. 6.1). Since the fingertip and the target are reduced to two virtual points in the Reacher-Extreme task, they are invisible in Fig. 6.1c.

For the non-visual task, the observation includes the position of the fingertip, the speed of the fingertip, and the vector from the fingertip to the target. For the visual task, we remove the vector from the fingertip to the target from the observation space since this information should be inferred from images. Because the visual task is intrinsically non-markovian, we stack three consecutive images of the size $70 \times 100 \times 3$ to provide histories to the agent. The action space is the torques to be applied to the two joints, scaled to the range -1 to 1. The reward function is modified to give -1 for each step to encourage shorter episodes. After each 1000 steps, we reset the agent by moving the fin-

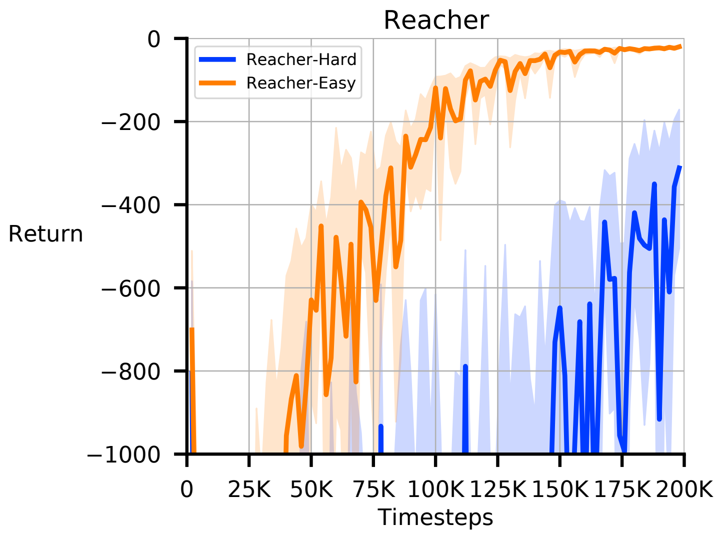


Figure 6.2: Performance of SAC on the minimum-time formulation of the two Reacher sub-tasks. Each solid curve is an average of 30 independent runs. The shaded regions represent a 95% confidence interval.

gertip to a random location on the plane while keeping the target unchanged. This process continues until the fingertip reaches the target within the target size. Once the target is reached, the current episode terminates. At episode terminations, we reset the agent and randomly generate a new target for the next episode.

6.2.2 Learning Performance

As shown in Fig. 6.2, SAC can solve the minimum-time Reacher-Easy task effectively within 200k steps. While there are signs of learning with the Reacher-Hard task, the learned policy after 200K timesteps isn't quite as successful. One possible explanation is that the initial policy $\mathcal{N}(0, 1)$ can stumble upon the target more frequently in Reacher-Easy compared to Reacher-Hard.

To test the explanation, we used SAC to solve the non-visual Reacher-Extreme task, whose target is virtually a point on the plane. Since the Reacher-Extreme task is significantly more difficult than Reacher-Hard, we suspect that no learning will happen for this task. As expected, SAC failed to learn an effective policy on Reacher-Extreme. There were only one or two episodes finished within the 200k training steps for all 30 seeds. As a result, no meaningful learning curves can be plotted. At least, the preliminary results confirm that SAC can reliably solve complex vision-based tasks in the minimum-time formulation. Based on the results, we define the difficulty of a minimum-time task by counting the number of target hits with the initial policy $\mathcal{N}(0, 1)$ exploration in 20k steps.

6.3 Time Limit as a Solution Parameter

We then investigated what factors would impact the difficulty of a minimum-time task. One evident factor is the exploration strategy. As discussed in Section 2.6, many works have proposed novel methods to improve exploration in sparse reward tasks in an effort to increase the chances of encountering the terminal state. In this thesis, we focus on an oft-ignored parameter in task specification, the time limit. Existing minimum-time tasks, such as Mountain Car (Moore, 1990; Sutton & Barto, 2018), use fixed time limits to improve exploration efficiency. Intuitively, a large time limit will increase the chance of reaching the target since it gives the agent more time to explore the environment. However, a large time limit will only hurt the exploration if the agent gets stuck in uninformative states due to a sub-optimal policy. In essence, the choice of time limit could drastically impact the frequency of reaching the terminal state. Thus, the optimal time limit should be a task-dependent parameter.

Moreover, time limits play a significant role in task design as well. For example, most OpenAI Gym environments treat time limits as an intrinsic property of their task formulations. In such environments, time limits are often used to separate episodes. That is, those environments define an episode as: $\{(S_{t-1}, A_t, S_t) | 1 \leq t \leq T\}$, where S_0 is the initial state of the episode and T=*Time Limit*.

We hypothesize that the choice of time limit can directly affect overall learning performance, as there exists a direct correlation between the frequency of reaching the terminal state and the final learning performance. Since our definition of an episode includes reset steps and is independent of the choice of time limit in the minimum-time formulation, we can safely modify the time limit of a task without altering the problem formulation. In short, **we treat the time limit as a tunable parameter that a learning system can tweak.** Using our difficulty definition, we evaluate the difficulties of the minimum-time Ball-in-Cup and Reacher for a set of time limits $\{1, 2, 5, 10, 25, 50, 100, 500, 1000, 5000\}$. Finally, based on the performance of SAC under different difficulties, we propose guidelines that practitioners can use to predict the performance of SAC on their own minimum-time tasks before the actual training.

Chapter 7

Minimum-Time Experiments

In this chapter, we show and discuss the results of the minimum-time experiments. Most importantly, we propose guidelines to determine if a minimum-time task is solvable by SAC before the actual training in this chapter.

7.1 Minimum-Time Ball-in-Cup Task

The tasks used in this chapter are Reacher and Ball-in-Cup. The Reacher task has been described in the previous chapter. The minimum-time Ball-in-Cup task was derived from the Deepmind Control Suite (Tassa et al., 2018). The original Ball-in-Cup is designed with the conventional sparse-reward formulation in which the agent receives a reward of 1 once it reaches the terminal state. We converted them into the minimum-time formulation, i.e., a reward of -1 will be given to the agent each timestep until termination. Besides the necessary changes to the reward functions and termination conditions, we also modified the task formulations to reflect typical setups of real robot tasks. Minimum-time Ball-in-Cup has two variants: visual and non-visual tasks. Visual tasks require the agent to learn useful information from images, which are more challenging than non-visual tasks.

This task aims to put a small ball into a receptacle. The ball is attached to the receptacle with an elastic string. Fig. 7.1 shows a screenshot of this task. The receptacle moves in the vertical plane (x-z plane) to swing and catch the ball.

For the non-visual task, the observation includes the ball's position and

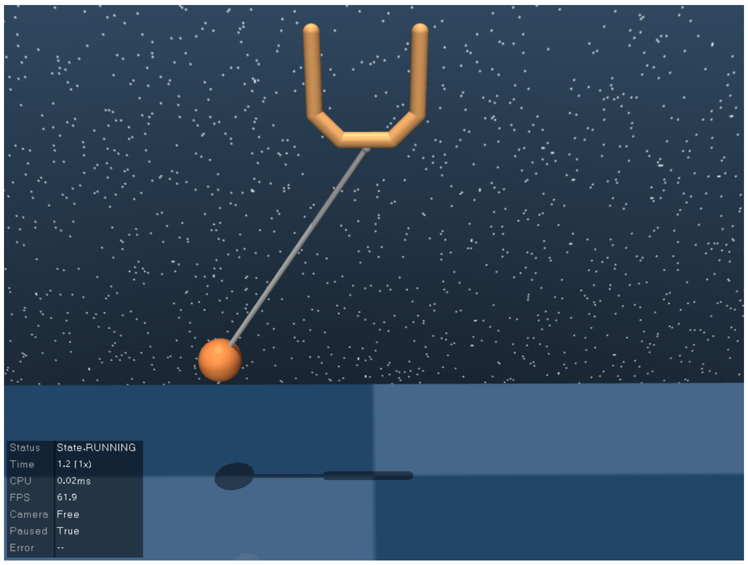
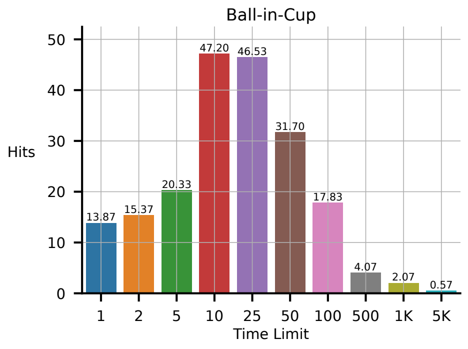


Figure 7.1: Ball in Cup Task from Deepmind Control Suite

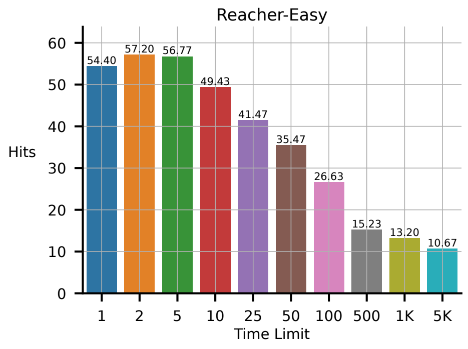
velocity and the receptacle's position. For the visual task, we remove both the ball and receptacle positions from the observation space since this information should be inferred from images. Because the visual task is intrinsically non-markovian, we stack three consecutive images of the size $100\times120\times3$ to provide histories to the agent. The action space is the force applied to the receptacle, scaled from -1 to 1. The reward function is modified to give -1 for each step to encourage shorter episodes. Once the receptacle catches the ball, the current episode terminates. The reset function moves the receptacle to a predefined location and the ball to a random location on the vertical plane upon timeout or episode terminations.

7.2 Relationship Between Time Limit & Learning Performance

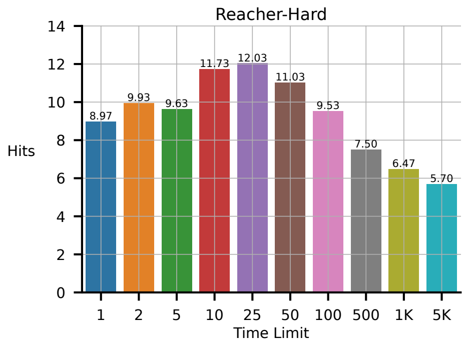
We plotted the number of target hits versus time limits for Ball-in-Cup and Reacher with the initial policy $\mathcal{N}(0,1)$ in Fig. 7.2. It reveals that time limits indeed affect an agent's exploration. It also demonstrates that the optimal time limit is task-dependent. Note that Reacher-Extreme has significantly fewer target hits than Reacher-Easy and Reacher-Hard for all time limits. Fig. 7.2 applies to both visual and non-visual tasks since their initial policies



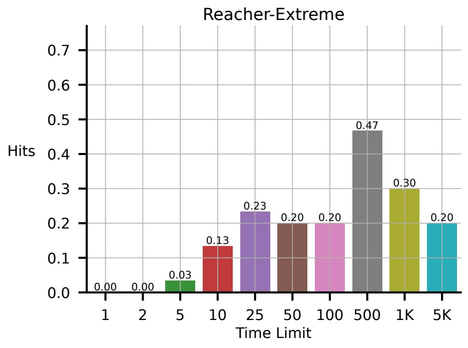
(a)



(b)



(c)



(d)

Figure 7.2: A histogram plot of the choice of time limit versus the number of target hits, that is, the number of times the agent reaches the terminal state using an initial policy within 20K timesteps. The error bar indicates the standard error estimated over 30 seeds.

and termination conditions are the same and independent of images.

Then we tested whether the number of target hits affects training performance. The tasks we used for this purpose are Non-Visual and Visual Ball-in-Cup, Non-Visual Reacher-Easy, and Non-Visual Reacher-Hard. The learning curves of each task across multiple time limits are shown in Fig. 7.3. It shows a clear relationship between the learning performance and the num-

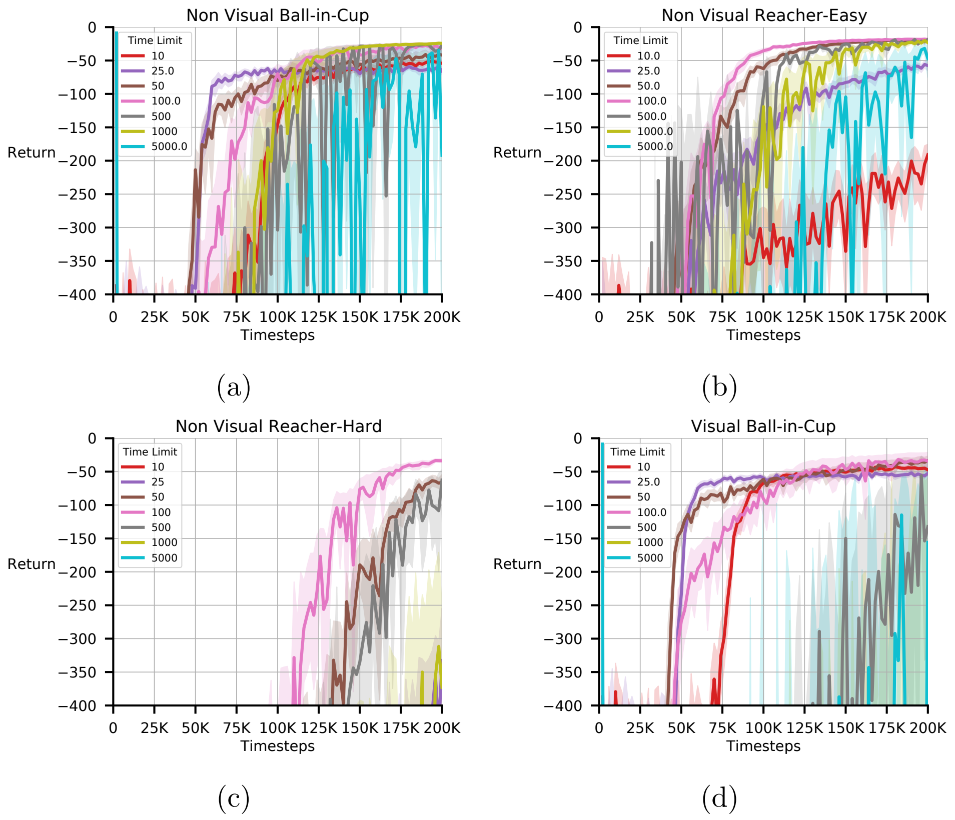


Figure 7.3: Learning performance of SAC on three non-visual and one visual task in simulation for multiple choices of the time limit. Each solid curve is averaged over 30 independent runs. The shaded regions represent a 95% confidence interval.

ber of target hits. Since the agent of Reacher-Extreme failed to reach the target during training for all time limits, no meaningful learning curves can be plotted here. Thus, the results **confirm our hypothesis** that the choice of time limit can directly affect overall learning performance, as there is a direct correlation between the frequency of reaching the terminal state and the final learning performance.

7.3 Guidelines for Effective and Robust Learning of Minimum-Time Tasks

Although all the time limits lead to successful learning, the learning rate and stability differ. Our simulation results show that short time limits give early success. However, using them for learning runs the risk of achieving a suboptimal policy if the minimum time to reach the terminal state is longer than the time limit. Therefore, it seems preferable to choose a long time limit. Unfortunately, Fig. 7.2 reveals that long time limits may also reduce the chance of reaching the target, which means learning can also be unreliable or even unsuccessful in a given amount of time, as is evident from the learning results.

Based on our observations, one strategy for time limit selection is to choose a time limit as small as possible as long as there is a sufficient number of hits and it is larger than the expected minimum steps to reach the target. Since the terminal state is defined to have a state value of 0, a sufficient number of target hits will give the agent enough information to learn the state values of other states along the same episode so that policy improvement can happen. If there are not enough hits, we can modify the time limit and measure the initial performance again. Once we have sufficient hits, we can attempt learning with that time limit. For example, time limits 1, 2, 5, 10, and 25 are not suitable for our simulation tasks since the minimum steps to the terminal state are about 25 to 30 for each task.

Based on our simulation results, we consider 10 to be a sufficient number for the average target hits per $20K$ steps. Hence, it allows for 50 hits on expectation in a replay buffer of $100K$ steps. For example, according to this heuristic, we can try a time limit of 100 for Ball-in-Cup, a time limit of 50 for Reacher-Easy, and a time limit of 50 for Reacher-Hard. Our learning curves show that the agent can learn with each choice of time limit. One caveat is that the chosen time limit is not necessarily the optimal time limit to solve the task.

To summarize, our guidelines for solving a minimum-time task are:

- Initialize the weights of the last layer of the policy network to all zeros so that the initial policy is the Gaussian distribution $\mathcal{N}(0, 1)$.

- Collect the number of target hits for a set of time limits using the initial policy.
- Select the time limits under which the number of target hits is larger than 10. The selected time limits should be larger than the expected minimum steps to reach the terminal state.
- Start from the smallest selected time limit and use our chosen hyperparameters and network initialization to train the task with SAC.

We will show the tests of the guidelines in the next chapter.

Chapter 8

Tests of Guidelines for Minimum-Time Tasks

We tested our guidelines by following them on seven held-out tasks: four real robot tasks and three simulation tasks. The three simulation tasks are visual Reacher-Easy, visual Reacher-Hard, and non-visual Dot-Reacher. The visual Reacher tasks are described in the previous chapters and the non-visual Dot-Reacher task will be described in this chapter. The four real robot tasks are Create-Reacher, UR5-Min-Time-Reacher, Vector-ChargerDetector, and Franka-Min-Time-Reacher. Details are given in the next section.

8.1 Tasks

We aim to investigate real-time learning of minimum-time real robotic tasks. For this purpose, we developed three real robot tasks with three robots and reused the Create-Reacher task. Unlike simulation tasks, all robotic tasks except Franka-VisualReacher require SenseAct (Rupam et al., 2018a) to run. Task details are given below.

8.1.1 UR5-VisaulReacher-MinTime

This task is the minimum-time formulation of UR5-VisaulReacher. We especially changed the reward function to -1 for each timestep to convert it into a minimum-time task. The modified task aims to move its fingertip to the target (red blob) on the screen as soon as possible. The current episode terminates once the target occupies more than 1.5% pixels of the current image. If the current episode does not terminate within the time limit, the reset function will move the arm to a predefined posture. During a reset, the target location remains unchanged. Like Reacher, once an episode terminates, we reset the arm and generate new random targets for the next episode. We also increased the maximum joint speed and movement bounds for better explorations.

8.1.2 Franka-VisualReacher

The Franka Emika Panda robot is a 7-DOF robot arm with a 3kg max payload and full torque sensing at each joint. Similar to the UR5, we do not control the arm's end-effector, and a Logitech Camera is attached to the tip of the arm. The agent controls the arm by sending velocity commands for each joint at 25Hz. The agent's task is to move the camera close to a randomly placed bean bag on the table using the images taken from the attached RGB camera. Fig. 8.1 shows the task setup. We provide the agent with three consecutive

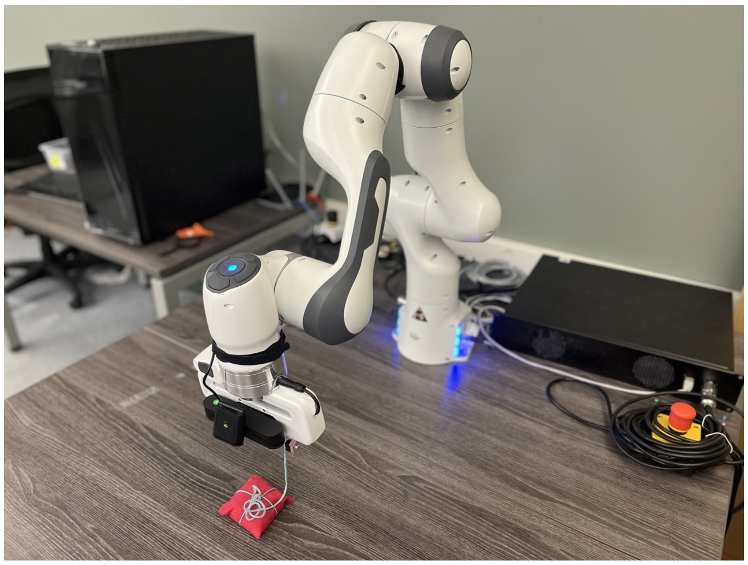


Figure 8.1: Franka VisualReacher

images of the size $160 \times 90 \times 3$ for a better sense of direction. We also provide the agent with the current positions of the joints, current joint velocities, and the previous velocity commands. The agent receives this information at 25Hz. At the beginning of an episode, we place the arm and the bean bag in random positions. The reward is -1 at every step until the episode ends. The episode ends when the bean bag covers more than 12% of the image captured by the camera. If the agent does not reach the bean bag within the time limit, we reset the environment by putting the bean bag and the arm at random positions.

8.1.3 Vector-ChargerDetector

Anki Vector is a low-cost mobile home robot approximately 10 cm in length and augmented with a 3D-printed bulldozer-like end-effector. It has two controllable wheels, a tiltable head with an LCD screen and camera, and an arm that can be used to lift or flip a cube. It also has a proximity sensor, four cliff sensors to detect falling edges, IMU for orientation and position tracking, and encoders to provide feedback on motor rotation. In this thesis, Vector is configured to use velocity control over WiFi.

We propose a novel vision-based benchmark task called Vector-ChargerDetector. Fig. 8.2 shows the task setup. The observation space consists of four stacked

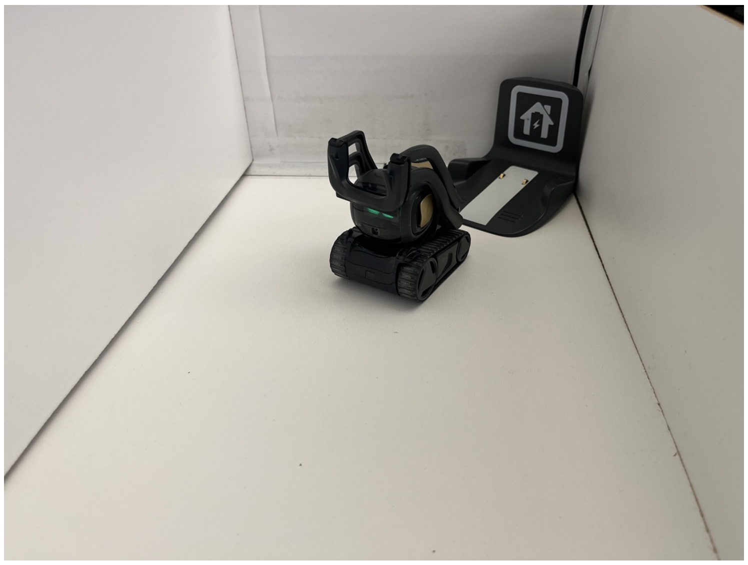


Figure 8.2: Vector ChargerDetector

$160 \times 120 \times 3$ images to give the learning agent temporal history, proximity sensor values, wheel velocities, and the previous action. The agent receives a new image every $200ms$, and the robot state information every $60ms$. The agent sends an action to the robot every $100ms$ over WiFi. The episode terminates when the charger symbol is centered and occupies roughly 25% of the image. Our reset routine brings Vector to a random location in the arena if the current episode does not terminate within the time limit. During a reset, Vector first moves backward and reorients itself in a random direction. Our reset routine can also handle re-charging when the battery level falls below a specified threshold and recovering from a failure mode like flipping over.

8.1.4 Minimum-Time Dot-Reacher

We also developed a simple environment called Minimum-Time Dot-Reacher with NumPy and Python as an additional simulation task for testing our guidelines. In this task, a blue dot tries to reach a stationary red target in a 2D plane (see Fig. 8.3). In addition, the blue dot has to reach the target within a specific velocity threshold. The observation space consists of the position

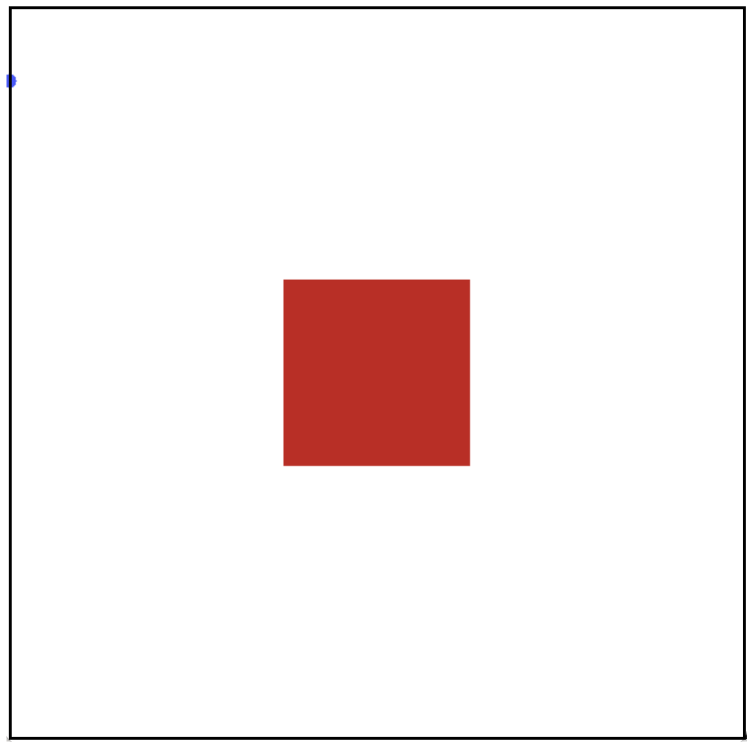

(a) Dot-Reacher

Figure 8.3: Dot-Reacher developed with Numpy and Python. The red area represents the target, and the blue dot represents the agent

and velocity of the blue dot, and the action space is the acceleration applied to the blue dot in the 2D plane, scaled to the range -1 to 1. At the start of a new episode, the blue dot starts in a random location of the 2D plane, excluding the target region. We use only the non-visual variant of this task in our experiments.

8.2 Experiment Setup

For simulation tasks, we tested three time limits, 50, 100, and 500, to see if learning could happen for at least one of the time limits. For each time limit and task, 30 runs were performed. But data collection is expensive and time-consuming for real robot tasks. Hence, we only tested three time limits on Franka-Min-Time-Reacher and heuristically chose one time limit for the other tasks. We deliberately chose the time limits for each robot task to have enough target hits with the initial policy. The time limit selection is summarized in Table 8.1. For Franka-Min-Time-Reacher, we performed five runs. For Create-Reacher, we performed three runs since the hardware was worn out after three complete runs. Due to the time limit and unstable WiFi connection, only two runs were performed for Vector-ChargerDetector. We used the ReLoD system to facilitate effective learning on all robot tasks.

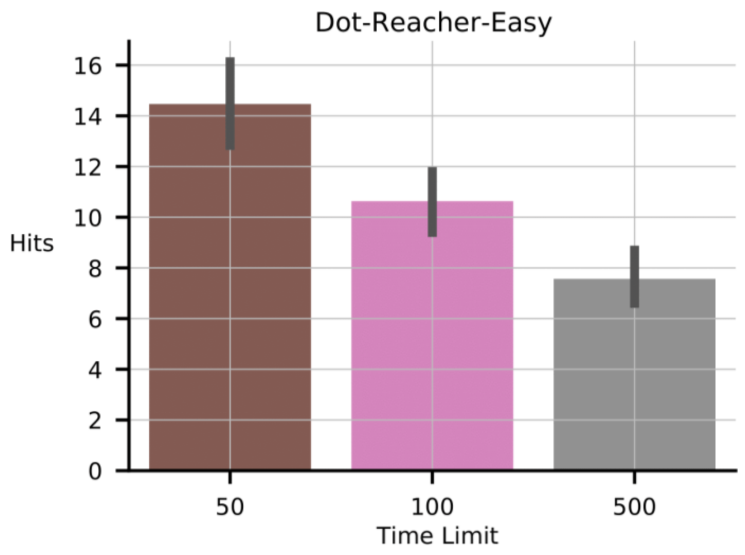
8.3 Results

In this section, we show and discuss the results of the tests of our guidelines for minimum-time tasks. Most importantly, our results confirm the usefulness of our guidelines in practice. [1]

8.3.1 Results on Simulation Tasks

The number of target hits versus time limits for Dot-Reacher is shown in Fig. 8.4. The same plot for Reacher and Ball-in-Cup can be found in Fig. 7.2.

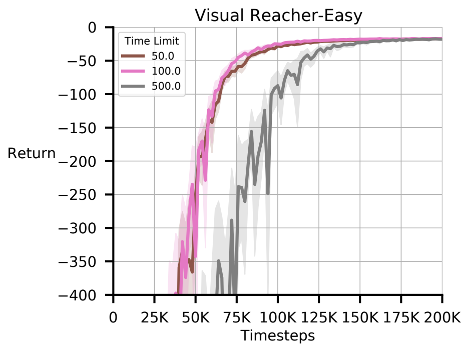
[1]Real robot experiment videos can be found at `https://sites.google.com/view/minimum-time-rl`



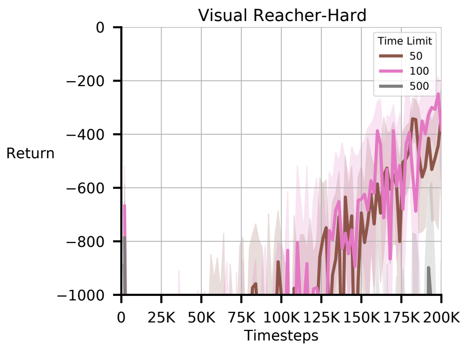
(a)

Figure 8.4: A histogram plot of the choice of time limit versus the number of target hits using an initial policy within 20K timesteps for Dot-Reacher. The error bar indicates the standard error estimated over 30 seeds.

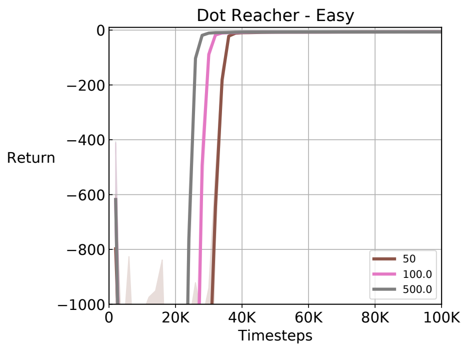
Fig. 8.4 and Fig. 7.2 show that all simulation tasks used for testing have enough target hits for at least one of the three time limits, which means learning should happen for at least one of the three time limits.



(a)



(b)



(c)

Figure 8.5: Learning performance of SAC on three held-out simulation tasks for three choices of time limit. Each solid curve is averaged over 30 independent runs. The shaded regions represent a 95% confidence interval.

We observed successful learning of all four simulation test tasks in Fig. 8.5., though the visual variants have slower and more variant learning curves. Fig. 8.5 **confirms** the usefulness of our guidelines in predicting the learning of simulation minimum-time tasks.

8.3.2 Results of Real Robot Tasks

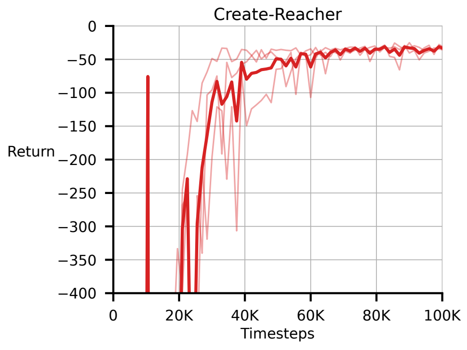
The time limits and the target hits for each robot are summarized in Table 8.1.

Task	Time Limit(s)	Average Target Hits	Training Steps	Wall Clock Time
Franka-Min-Time-Reacher	3s, 6s, 30s	12.6 ± 1.0, 13.0 ± 2.08, 7.8 ± 0.87	60k	2h $\sim$ 3h
Create-Reacher	15s	18.8 ± 3.92	100k	5h $\sim$ 7h
UR5-Min-Time-Reacher	6s	14.4 ± 2.5	100k	3h $\sim$ 4h
Vector-ChargerDetector	30s	11 ± 1.1	160k	16h $\sim$ 24h

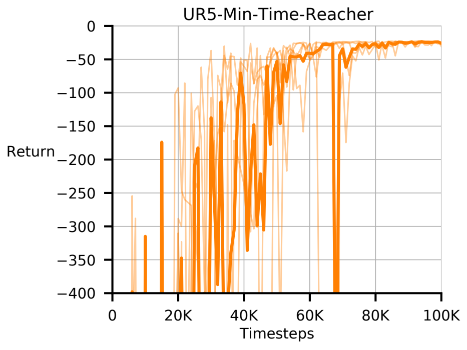
Table 8.1: Robot experiment setup details. The average target hits are calculated using 20K samples of real-world data for each task. The standard errors for the average target hits are estimated using five independent runs.

From Table 8.1, we can see that Franka-Min-TimeReacher and Vector-ChargerDetector have just about sufficient target hits. Create-Reacher and UR5-Min-Time-Reacher have more than enough target hits. According to our guidelines, SAC should be able to learn effective policies on all tasks.

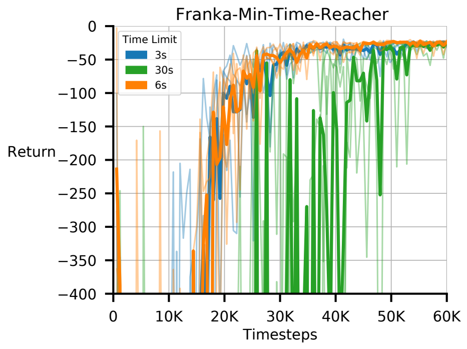
8.6.



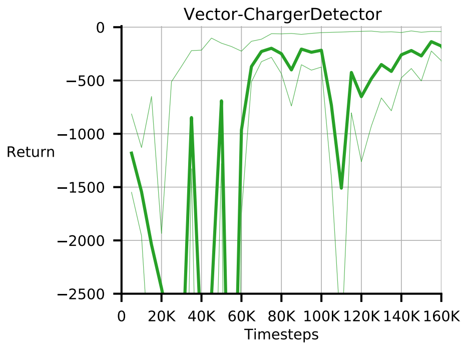
(a)



(b)



(c)



(d)

Figure 8.6: Learning performance of SAC on four vision-based policy learning tasks with robots. Each thin learning curve here represents an independent run. The bold learning curves of the same color are the average of all independent runs.

The volatility in the learning curves of Vector-ChargerDetector is due to the variable WiFi latency. This problem is inherent to Vector because Vector's WiFi adapter struggles to establish a stable WiFi connection with our WiFi router. The successful learning curves of the four robot tasks shown in Fig. 8.6 **confirm** the usefulness of our guidelines in predicting the learning performance of real robot minimum-time tasks.

Chapter 9

Conclusion

In this thesis, we empirically investigated two oft-ignored issues of real-time learning of robotic tasks: 1) How limited computation resources impact the performance of a learning system and how to use a wirelessly connected powerful computer to compensate for the performance loss; 2) how to design reward functions for vision-based real-time learning tasks.

To address the first issue, we introduced the ReLoD system for learning to control robots by a real-time RL agent that distributes its computations between a local and a remote computer. In our experiments with ReLoD, we demonstrated that when SAC ran on a resource-limited computer, its performance could be dramatically reduced. Moreover, solely deploying all computations of SAC on a wirelessly connected remote workstation may not improve performance due to latency. On the other hand, our results suggest that performing local action computation and remote policy updating compensates substantially for SAC's performance loss. However, this distribution may not benefit all RL algorithms since PPO's performance was nearly unaffected by the distribution mode. In addition, ReLoD's Local-Only mode is suitable for conventional RL research as its performance was shown to be on par with a well-tuned single-computer system. Due to the latency in communicating actions, ReLoD's Remote-Only mode should only be used when the local computer cannot compute actions within the action cycle time. We conducted 60 independent runs in our experiments, which took nearly 185 hours of usage on real robots.

For the second issue, we focused on effectively solving minimum-time tasks with a reward of -1 for each step. Through an empirical investigation of the performance of SAC on multiple complex vision-based simulations and real robotic control tasks in the minimum-time formulation, we identified that an agent could achieve successful learning performance if it can reach terminal states often enough using its initial policy. Contrary to popular belief, we showed that it is possible to have robust, reliable learning from scratch on complex, vision-based robotic control tasks using only sparse rewards. We established that the time limit should be a tunable solution parameter instead of a problem parameter in the minimum-time formulation. We also outlined useful guidelines that practitioners can use to determine if the minimum-time formulation is suitable for their tasks and described the conditions under which they can be solved by SAC effectively. Our work is the first demonstration of using a single reinforcement learning system to achieve real-time learning of pixel-based control of multiple physical robots from scratch in the minimum-time formulation. In total, we conducted 1200 independent runs in simulation and 28 independent runs in the real world, which took about 340 hours.

9.1 Limitations

This work has several limitations. First, although we gave a possible explanation as to why the remote-only mode of the SAC-agent performs worse than the remote-local mode in the first part, we still do not fully understand how the remote-local mode reduces action latency. Second, the learning curves of PPO in the first part are not plateau. Thus, more steps are needed to test PPO's performance. Third, in the first part, we only implemented SAC and PPO and tested them on two tasks, which may limit the generalizability of our results to other tasks. Fourth, we only tested our guidelines on seven tasks in the second part. Our guidelines could be more reliable if they were tested on more tasks. And finally, in the second part, we performed a linear search for the time limit parameter, which can be inefficient.

9.2 Future Work

A natural extension to the first part of this work is to study how the remote-local mode reduces action latency in wireless networks. We could decompose action latency into multiple delay components and evaluate which components impact performance significantly. Then a detailed investigation can be carried out to evaluate how and to what extent remote-local mode reduces those delays.

In the second part of this work, we showed that the performance of the initial policy is important to solve minimum-time tasks. However, the performance of the initial policy depends on how the weights of the policy network are initialized. Studying how different initialization methods impact learning performance is interesting. Furthermore, since we treat the time limit as a solution parameter in the minimum-time setting, it will be more efficient to search for the optimal time limit with Bayesian optimization or Luby search instead of linear search.

References

Abdolmaleki, A., Springenberg, J. T., Tassa, Y., Munos, R., Heess, N., & Riedmiller, M. (2018, June 14). Maximum a posteriori policy optimisation. *arXiv preprint arXiv:1806.06920.*

Achiam, J. (2018). *Spinning Up in Deep Reinforcement Learning.* GitHub.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Pieter Abbeel, O. A. I., & Zaremba, W. (1970, January 1). Hindsight experience replay. *Advances in Neural Information Processing Systems, 30.*

Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., ... & Zhang, S. (2019). Dota 2 with large-scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680.*

Bloesch, M., Humplik, J., Patraucean, V., Hafner, R., Haarnoja, T., Byravan, A., ... & Heess, N. (2022, January). Towards real robot learning in the wild: A case study in bipedal locomotion. In *Conference on Robot Learning* (pp. 1502-1511). PMLR.

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., & Zieba, K. (2016, April 25). End-to-end learning for self-driving cars *arXiv preprint arXiv:1604.07316.*

Brockman, G., Zaremba, W., Tang, J., Schulman, J., Schneider, J., Pettersson, L., & Cheung, V. (2016). *Openai/gym: A toolkit for developing and comparing reinforcement learning algorithms.* GitHub.

Chaumette, F., & Hutchinson, S. (2006). Visual servo control. I. Basic Approaches. *IEEE Robotics & Automation Magazine, 13*(4), 82–90.

Codevilla, F., Muller, M., Lopez, A., Koltun, V., & Dosovitskiy, A. (2018).

End-to-end driving via conditional imitation learning. *2018 IEEE International Conference on Robotics and Automation (ICRA).*

Collewet, C., & Chaumette, F. (2002). Positioning a camera with respect to planar objects of unknown shape by coupling 2-D visual servoing and 3-D estimations. *IEEE Transactions on Robotics and Automation, 18*(3), 322–333.

Conkie, A., & Chongstitvatana, P. (1990). An uncalibrated stereo visual servo system. *Procedings of the British Machine Vision Conference 1990.*

Corke, P. I., & Hutchinson, S. A. (2001). A new partitioned approach to image-based visual servo control. *IEEE Transactions on Robotics and Automation, 17*(4), 507–515.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems, 2*(4), 303–314.

Ebert, F., Finn, C., Dasari, S., Xie, A., Lee, A., & Levine, S. (2018, December 3). Visual foresight: Model-based Deep Reinforcement Learning for vision-based robotic control *arXiv preprint arXiv:1812.00568.*

Finn, C., & Levine, S. (2017). Deep visual foresight for planning robot motion. *2017 IEEE International Conference on Robotics and Automation (ICRA).*

Finn, C., Xin Yu Tan, Yan Duan, Darrell, T., Levine, S., & Abbeel, P. (2016). Deep spatial autoencoders for Visuomotor Learning. *2016 IEEE International Conference on Robotics and Automation (ICRA).*

Fujimoto, S., Meger, D., & Precup, D. (2019, May). Off-policy deep reinforcement learning without exploration. In *International conference on machine learning* (pp. 2052-2062). PMLR.

Haarnoja, T., Ha, S., Zhou, A., Tan, J., Tucker, G., & Levine, S. (2019, June 19). Learning to walk via Deep Reinforcement Learning *arXiv preprint arXiv:1812.11103.*

Haarnoja, T., Tang, H., Abbeel, P., & Levine, S. (2017, July 17). Reinforcement learning with deep energy-based policies. In *International conference on machine learning* (pp. 1352-1361). PMLR.

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018, July 3). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning* (pp. 1861-1870). PMLR.

Hasselt, H. (2010). Double Q-learning. *Advances in Neural Information Processing Systems*, 23.

Hertweck, T., Riedmiller, M., Bloesch, M., Springenberg, J. T., Siegel, N., Wulfmeier, M., Hafner, R., & Heess, N. (2020, May 15). Simple sensor intentions for exploration. *arXiv preprint arXiv:2005.07541.*

James, S., & Johns, E. (2016, December 13). 3D simulation for robot arm control with Deep Q-Learning. *arXiv preprint arXiv:1609.03759.*

Jägersand, M. (1996). Visual servoing using trust-region methods and estimation of the full coupled visual-motor Jacobian. *Image (IN), 11,* 1.

Kahn, G., Abbeel, P., & Levine, S. (2021). Badgr: An autonomous self-supervised learning-based navigation system. *IEEE Robotics and Automation Letters, 6*(2), 1312-1319.

Kalashnikov, D., Irpan, A., Pastor, P., Ibarz, J., Herzog, A., Jang, E., ... & Levine, S. (2018, October). Scalable deep reinforcement learning for vision-based robotic manipulation. In *Conference on Robot Learning* (pp. 651-673). PMLR.

Kohl, N., & Stone, P. (2004). Policy gradient reinforcement learning for fast quadrupedal locomotion. *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004.*

Koivo, A. J., & Houshangi, N. (1991). Real-time vision feedback for servoing robotic manipulator with self-tuning controller. *IEEE Transactions on Systems, Man, and Cybernetics, 21*(1), 134–142.

Kostrikov, I., Yarats, D., & Fergus, R. (2021, March 7). Image augmentation is all you need: Regularizing deep reinforcement learning from pixels. *arXiv preprint arXiv:2004.13649.*

Krishnan, S., Boroujerdian, B., Fu, W., Faust, A., & Reddi, V. J. (2021). Air learning: A deep reinforcement learning gym for Autonomous Aerial Robot Visual navigation. *Machine Learning, 110*(9), 2501–2540.

Kumar, A., Fu, J., Soh, M., Tucker, G., & Levine, S. (2019). Stabilizing off-policy q-learning via bootstrapping error reduction. *Advances in Neural Information Processing Systems, 32.*

Kumar, A., Zhou, A., Tucker, G., & Levine, S. (2020). Conservative q-learning for offline reinforcement learning. *Advances in Neural Information Processing Systems, 33,* 1179-1191.

Lambert, N. O., Drew, D. S., Yaconelli, J., Levine, S., Calandra, R., & Pister, K. S. (2019). Low-level control of a quadrotor with deep model-based reinforcement learning. *IEEE Robotics and Automation Letters, 4*(4), 4224–4230.

Laskin, M., Lee, K., Stooke, A., Pinto, L., Abbeel, P., & Srinivas, A. (2020). Reinforcement learning with augmented data. *Advances in neural information processing systems, 33*, 19884-19895.

LeCun, Y., Haffner, P., Bottou, L., & Bengio, Y. (1999). Object recognition with gradient-based learning. In *Shape, contour and grouping in computer vision* (pp. 319-345). Springer, Berlin, Heidelberg.

Leshno, M., Lin, V. Y., Pinkus, A., & Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks, 6*(6), 861–867. Levine, S., Finn, C., Darrell, T., & Abbeel, P. (2016). End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research, 17*(1), 1334–1373.

Levine, S., Pastor, P., Krizhevsky, A., Ibarz, J., and Quillen, D. (2018). Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research, 37*(4-5): 421–436.

Li, J., Koyamada, S., Ye, Q., Liu, G., Wang, C., Yang, R., ... & Hon, H. W. (2020). Suphx: Mastering mahjong with deep reinforcement learning. *arXiv preprint arXiv:2003.13590.*

Li, J., Koyamada, S., Ye, Q., Liu, G., Wang, C., Yang, R., ... & Hon, H. W. (2020). Suphx: Mastering mahjong with deep reinforcement learning. *arXiv preprint arXiv:2003.13590.*

Liu, Q., Li, L., Tang, Z., & Zhou, D. (2018). Breaking the curse of horizon: Infinite-horizon off-policy estimation. *Advances in Neural Information*

Processing Systems, 31.

Malis, E. (2001). Hybrid vision-based robot control robust to large calibration errors on both intrinsic and extrinsic camera parameters. *2001 European Control Conference (ECC).*

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

Mahmood, A. R., Korenkevych, D., Komer, B. J., & Bergstra, J. (2018, October). Setting up a reinforcement learning task with a real-world robot. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (pp. 4635-4640). IEEE.

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, February 25). *Human-level control through deep reinforcement learning.* Nature News.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... & Kavukcuoglu, K. (2016, June). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*(pp. 1928-1937). PMLR.

Mo, K., Li, H., Lin, Z., & Lee, J. Y. (2018). The adobeindoornav dataset: Towards deep reinforcement learning based real-world indoor robot visual navigation. *arXiv preprint arXiv:1802.08824.*

Moore, A. W. (1990). *Efficient memory-based learning for robot control* (No. UCAM-CL-TR-209). University of Cambridge, Computer Laboratory.

Nair, A. V., Pong, V., Dalal, M., Bahl, S., Lin, S., & Levine, S. (2018). Visual reinforcement learning with imagined goals. *Advances in neural information processing systems, 31.*

Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., De Maria, A., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., Legg, S., Mnih, V., Kavukcuoglu, K., & Silver, D. (2015, July 16). Massively parallel methods for deep reinforcement learning. *arXiv preprint* *arXiv:1507.04296.*

Pervez, A., Mao, Y., & Lee, D. (2017). Learning deep movement primitives using convolutional neural networks. *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids).*

Pinkus, A. (1999). Approximation theory of the MLP model in Neural Networks. *Acta Numerica, 8,* 143–195.

Pinto, L. and Gupta, A. (2016). Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours. In *2016 IEEE international conference on robotics and automation (ICRA),* pages 3406–3413. IEEE.

Pomerleau, D. A. (1988). Alvinn: An autonomous land vehicle in a neural network. *Advances in Neural Information Processing Systems, 1.*

Ramstedt, S., & Pal, C. (2019). Real-Time Reinforcement Learning. *Advances in Neural Information Processing Systems, 32.*

Ravichandar, H., Polydoros, A. S., Chernova, S., & Billard, A. (2020). Recent Advances in Robot Learning from Demonstration. *Annual Review of Control, Robotics, and Autonomous Systems, 3,* 297–330.

Riedmiller, M., Hafner, R., Lampe, T., Neunert, M., Degrave, J., Wiele, T., ... & Springenberg, J. T. (2018, July). Learning by playing solving sparse reward tasks from scratch. In *International conference on machine learning* (pp. 4344-4353). PMLR.

Ross, S., Gordon, G., & Bagnell, D. (2011, June 14). A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (pp. 627-635). JMLR Workshop and Conference Proceedings.

Ross, S., Melik-Barkhudarov, N., Shankar, K. S., Wendel, A., Dey, D., Bagnell, J. A., & Hebert, M. (2013). Learning monocular reactive UAV control in cluttered natural environments. *2013 IEEE International Conference on Robotics and Automation.*

Rusu, A. A., Večerík, M., Rothörl, T., Heess, N., Pascanu, R., & Hadsell, R. (2017, October). Sim-to-real robot learning from pixels with progressive nets. In Conference on robot learning. In *Conference on robot learning* (pp. 262-270). PMLR.

Schulman, J., Levine, S., Moritz, P., Jordan, M., & Abbeel, P. (2015, July 1). Trust Region Policy Optimization. In *International conference on machine learning* (pp. 1889-1897). PMLR.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017, August 28). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347.*

Sharma, P., Pathak, D., & Gupta, A. (2019). Third-person visual imitation learning via decoupled hierarchical controller. *Advances in Neural Information Processing Systems, 32.*

Smith, L., Kostrikov, I., & Levine, S. (2022). A walk in the park: Learning to walk in 20 minutes with model-free reinforcement learning. *arXiv preprint arXiv:2208.07860.*

Sutton, R. S., Bach, F., & Barto, A. G. (2018). *Reinforcement learning: An introduction.* MIT Press Ltd.

Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (1999). Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems, 12.*

Tian, S., Nair, S., Ebert, F., Dasari, S., Eysenbach, B., Finn, C., & Levine, S. (2020, December 30). Model-based visual planning with self-supervised functional distances. *arXiv preprint arXiv:2012.15373.*

Tzeng, E., Devin, C., Hoffman, J., Finn, C., Abbeel, P., Levine, S., Saenko, K., & Darrell, T. (2020). Adapting deep visuomotor representations with weak pairwise constraints. *Springer Proceedings in Advanced Robotics,* 688–703.

Vasan, G., & Pilarski, P. M. (2017). Learning from demonstration: Teaching a myoelectric prosthesis with an intact limb via reinforcement learning. *2017 International Conference on Rehabilitation Robotics (ICORR).*

Vogt, D., Stepputtis, S., Grehl, S., Jung, B., & Ben Amor, H. (2017). A system for learning continuous human-robot interactions from human-human demonstrations. *2017 IEEE International Conference on Robotics and Automation (ICRA).*

Wahlström, N., Schön, T. B., & Deisenroth, M. P. (2015). Learning deep dynamical models from Image Pixels. *IFAC-PapersOnLine, 48*(28), 1059–1064.

Weiss, L., Sanderson, A., & Neuman, C. (1987). Dynamic sensor-based control of robots with visual feedback. *IEEE Journal on Robotics and Automation, 3*(5), 404–417.

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

Yoshimi, B. H., & Allen, P. K. (n.d.). Active, uncalibrated visual servoing. *Proceedings of the 1994 IEEE International Conference on Robotics and Automation.*

Young, S., Gandhi, D., Tulsiani, S., Gupta, A., Abbeel, P., & Pinto, L. (2021, October). Visual imitation made easy. In *Conference on Robot Learning* (pp. 1992-2005). PMLR.

Yuan, Y., & Mahmood, A. R. (2022). Asynchronous reinforcement learning for real-time control of Physical Robots. *2022 International Conference on Robotics and Automation (ICRA).*

Zhang, T., McCarthy, Z., Jow, O., Lee, D., Chen, X., Goldberg, K., & Abbeel, P. (2018). Deep imitation learning for complex manipulation tasks from virtual reality teleoperation. *2018 IEEE International Conference on Robotics and Automation (ICRA).*

Zhang, R., Dai, B., Li, L., & Schuurmans, D. (2020a). Gendice: Generalized offline estimation of stationary values. In *International Conference on Learning Representations.*

Zhang, S., Liu, B., and Whiteson, S. (2020b). Gradientdice: Rethinking generalized offline estimation of stationary values. *arXiv preprint arXiv:2001.11113.*

Zhu, Y., Mottaghi, R., Kolve, E., Lim, J. J., Gupta, A., Fei-Fei, L., & Farhadi, A. (2017). Target-driven visual navigation in indoor scenes using Deep Reinforcement Learning. *2017 IEEE International Conference on Robotics and Automation (ICRA).*

Zhu, Z., & Hu, H. (2018). Robot learning from demonstration in Robotic Assembly: A survey. *Robotics, 7*(2), 17