Robot Deep Reinforcement Learning: Tensor State-Action Spaces and Auxiliary Task Learning with Multiple State Representations

Devin Schwab

CMU-RI-TR-20-46

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Robotics

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

August 19, 2020

Thesis Committee:
Manuela Veloso, Chair
Katerina Fragkiadaki
David Held
Martin Riedmiller, Google DeepMind

Keywords: robotics, reinforcement learning, auxiliary task, multi-task learning, transfer learning, deep learning

To my friends and family.

Abstract

A long standing goal of robotics research is to create algorithms that can automatically learn complex control strategies from scratch. Part of the challenge of applying such algorithms to robots is the choice of representation. Reinforcement Learning (RL) algorithms have been successfully applied to many different robotic tasks such as the Ball-in-a-Cup task with a robot arm and various RoboCup robot soccer inspired domains. However, RL algorithms still suffer from issues of large training time and large amounts of required training data. Choosing appropriate representations for the state space, action space and policy can go a long way towards reducing the required training time and required training data.

This thesis focuses on robot deep reinforcement learning. Specifically, how choices of representation for state spaces, action spaces, and policies can reduce training time and sample complexity for robot learning tasks. In particular the focus is on two main areas:

1. Transferrable Representations via Tensor State-Action Spaces
2. Auxiliary Task Learning with Multiple State Representations

The first area explores methods for improving transfer of robot policies across environment changes. Learning a policy can be expensive, but if the policy can be transferred and reused across similar environments, the training costs can be amortized. Transfer learning is a well-studied area with multiple techniques. In this thesis we focus on designing a representation that makes for easy transfer. Our method maps state-spaces and action spaces to multi-dimensional tensors designed to remain a fixed dimension as the number of robots and other objects in an environment varies. We also present the Fully Convolutional Q-Network (FCQN) policy representation, a specialized network architecture that combined with the tensor representation allows for zero-shot transfer across environment sizes. We demonstrate such an approach on simulated single and multi-agent tasks inspired by RoboCup Small Size League (SSL) and a modified version of Atari Breakout. We also show that it is possible to use such a representation and simulation trained policies with real-world sensor data and robots.

The second area examines how strengths in one robot Deep RL state representation can make-up for weaknesses in another. For example, we would often like to learn tasks using the robot's available sensors, which include high-dimensional sensors such as cameras. Recent Deep RL algorithms can learn with images, but the amount of data can be prohibitive for real robots. Alternatively, one can create a state using a minimal set of features necessary for task completion. This has the advantages of 1) reducing the number of policy parameters and 2) removing irrelevant information. However, extracting these features often has a significant cost in terms of engineering, additional hardware, calibration and fragility outside the lab. We demonstrate this on multiple robot platforms and tasks in both simulation and the real-world. We show that it works on simulated RoboCup Small Size League (SSL) robots. We also demonstrate that such techniques allow for from scratch learning on real hardware via the Ball-in-a-Cup task performed by a robot arm.

Acknowledgments

I would like to thank my family. From an early age they instilled in me the value of a good education. I would never have pursued, let alone finished a PhD, without their upbringing and continued support throughout my life.

My husband, Tommy Hu, has also played an important role in my success. He has been a rock who has helped me get through difficult patches. Without his insights, love, and support I would not have been able to persevere and complete this thesis.

My advisor, Manuela Veloso, has also played a large role in my thesis. I could not have completed my thesis without the work with the RoboCup Small Size League soccer robots she helped to start. I would also like to thank her for giving me the freedom throughout my PhD to work on what interested me most.

I want to thank Martin Riedmiller, a member of my committee and my supervisor during the work I did at DeepMind. Without his support and the idea to work on the Ball-in-a-Cup task, a large portion of this thesis might not have happened. I would also like to thank the others I worked with at DeepMind both on the controls team and in the robotics lab. It was incredibly helpful bouncing ideas off of them and learning from their experiences.

I also would like to thank the other two members of my committee: David Held and Katerina Fragkiadaki. I enjoyed my research talks with David in the last few years of the PhD. His insights were extremely valuable. Katerina gave me the opportunity to TA for the new Reinforcement Learning class she was co-teaching. That was an amazing experience for me. I thoroughly enjoyed planning the curriculum and assignments with her and getting to introduce others at CMU to the field of Reinforcement Learning.

I would like to thank my master degree advisor, Soumya Ray. Without him I would not have been involved in Reinforcement Learning at all. His undergraduate AI class caught my imagination and focused my ambition towards AI and Reinforcement Learning. He then guided me through my masters degree and my first major publication. His support was invaluable when applying to PhD programs. Without his recommendation letters and advice crafting the applications, I doubt I would have been able to complete a CMU PhD.

I would also like to thank The University of Hong Kong and my colleagues that worked on the DARPA Robotics Challenge. It was an extremely useful experience for me, giving me my first taste of real robotics research. It was also incredibly rewarding to get to personally work with the Atlas robot.

Finally, I would like to thank my CORAL lab mates and other RI students, both past and present. Those who came before me in the CORAL lab and the RI PhD program had invaluable advice about how to navigate a PhD. And those who went through the process with me were always quick to help when I was stuck on a problem or just needed some support. I want to give special thanks to past members of the CMDragons SSL team, my thesis was built on the shoulders of their previous work.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Learning algorithms that allow robots to learn complex control strategies are a key component to making robots more useful and ubiquitous in everyday society. Reinforcement Learning (RL) research has made great progress towards making this vision a reality, having been successfully applied to many complex real-world domains such as: learning to perform Ball-in-a-Cup task with a robot arm, playing RoboCup robot soccer inspired games, manipulating objects, and flying a model helicopter. [2, 79, 87, 89, 116] However, despite recent advancement in Deep RL, there are still scalability issues due to the large amount of training time and training data required. Solving this entire problem is an undertaking too large for a single thesis. While there are many avenues of research investigating solutions for the scalability issues such as improvements to the policy learning algorithm and intelligent exploration methods, we focus on exploring how *choices in representation for the states, actions and policies* can help tackle these issues.

In this thesis, we focus on exploring how representations affect robot deep reinforcement learning and how to choose good representations for states, actions, and policies, such that the burden of training time and training data can be reduced. Choosing a good representation can be the difference between a robust robot policy that works outside the lab, or a fragile robot policy that works only in highly controlled environments. For example, a feature-based representation that requires a motion capture system to provide high accuracy position features will be excellent for learning a policy in the lab. However, it will be difficult to utilize such a policy anywhere this motion capture equipment is unavailable. Alternatively, other sensor types such as images or range finders can be utilized. However, these types of representations often require large amounts of training data in order to converge to reasonable behaviors, as the information important to the control task is not as distilled as in the feature-based representation. Which choice is made has a large effect on the capabilities of the robot system and how fast and efficient the learning can be.

Representations can also affect how well a robot policy generalizes to environments that differ from the training environment. The more generalizable a policy, the less likely additional training data and time will be required to make adjustments for the environment changes. If the types of changes can be anticipated, it is possible to design representations that increase a policies generalizability across these changes. Even if the up-front training cost is high, if the

policies can be applied across many similar environments, these costs can be amortized. For example, using highly distilled feature-based representations may allow for efficient learning on one task, but if care is not taken, the system may be locked into the specifics of the original task environment. On the other hand, one may choose to use highly general image sensors, which contain a large amount of information and work in general environments. However, these sensor types can over-fit to the training data. When picking representations for RL on robotic systems, one must be careful to find the sweet-spot between fast training time and generalizability.

Finally, representations of a policy have a large effect in encoding priors about how states and actions are processed and what types of behaviors are representable. For example, convolutional layers encode an assumption of translational invariance and that locality is important for processing the input. If the representation requires a highly non-local view of input or the inductive bias of translational invariance is a bad fit for a problem, then different layer types might be required.

Naturally, each of these components have been explored in some capacity, see Chapter 5 for a discussion on existing work in this area. However, in this work we focus specifically on tasks involving robot systems. Additionally, we focus on the representation choices that can be combined with other approaches to scalability like better exploration methods and policy learning algorithms.

1.2 Thesis Question

Given the large number of choices in representation available for any RL application, this thesis tries to address the following question:

> *How can we improve **scalability, transferrability, and deployability** of robot deep reinforcement learning via intelligent choices of **state, action, and policy network representations** where: environment sizes change, number of robots vary, states and actions are high-dimensional, and tasks have sparse rewards.*

By **scalability** we refer to the amount of training time and data required, as we believe this is one of the biggest blockers to learning with real robot hardware. Unlike in simulation, when training on real hardware, the world cannot be sped-up to train faster. The only potential speed-up is to parallelize the training with multiple robots in multiple training environments. However, given the current day cost of robots, building the necessary robots and the necessary training environments can be prohibitively expensive if a large amount of training data is required. Additionally, when training on real-hardware, one must be concerned with the safety of the robot and the environment. The faster the robot converges to reasonable behavior, the less time there is for catastrophic mistakes to be made.

There are many dimensions of **transferrability**, that can be considered, but in this thesis we focus on transfer as the number of robots varies, the number of other objects in the environment vary, and the size of the environments change. Transferrability of a policy is important for real-robot systems because it can help a policy learned in a simplified lab environment become applicable to similar, but larger, or more complex real-world environments. Additionally, even if training costs are high initially, the more transferrable a policy is, the more this cost can be

amortized across different changes to the environment.

Finally, we focus on **deployability** of our policies to real hardware. Whether the policy is trained entirely in simulation or directly on the real-robot hardware. We want to design methods that lead to policies that at test time work on the real hardware. Without special care, simulations can ignore crucial difficulties with real-world robot RL, such as safe execution, the asynchronous nature of the environment, and the real-time requirements for running a policy in dynamic environments. This can lead to policies that train and test well in simulation, but completely fail during tests on real hardware. In this thesis we focus on methods that at test time can run on real-hardware, regardless of if the training was done in simulation or real-hardware.

We address aspects of all these concerns through a combination of techniques involving choices of representations in the states, actions, and policy networks. By state and action representations we refer to aspects such sensor types and actuator types, along with the policy's input state-space and output action space, which may be different from real-robot high-dimensional sensors. We also consider how specific network architectures combine with these state-action representations, such that we can leverage data sharing across modalities or build in biases that help speed-up training in these spaces.

1.3 Approach

As shown in the previous section, there are a large number of choices to make when designing a representation for a robot RL task. This thesis focuses on two different aspects of representations used in robot RL:

1. Transferrable Representations via Tensor State-Action Spaces
2. Auxiliary Task Learning with Multiple State Representations

The first approach explores how representations can be designed specifically with transfer in mind. While the initial learning may be expensive, if the policy can be transferred across a variety of environment changes, this training time can be amortized across many scenarios.

The second approach examines how weaknesses in a test time representation can be compensated for at training time by other representations. The goal being to end with a flexible, robust representation, while speeding up the training by utilizing information from alternative representations. This can be especially useful for learning policies in high-dimensional state-action spaces.

1.3.1 Transferrable Representations

A few approaches to transfer learning exist. This thesis specifically focuses on designing representations that can be transferred across specific types of changes to the environment. By picking a representation with transfer in mind, it is possible to include biases about how the policy should adapt to environment changes. It also can remove the need to learn or adapt weights after a change has occurred. The advantage of a transferrable representation is that even if the learning takes a large amount of time and training data initially, this cost can be amortized by allowing a policy to apply to a wide variety of similar environments. We look at not only transferring

single agent policies where the number objects in the environment are changing, but transferring multi-agent policies where the number of agents in the environment may vary.

We additionally, present the Fully Convolutional Q-Network (FCQN) policy network representation, which is designed to work with our transferrable representation in environments where robot and object positioning is important to task performance. In such environments, policy training time can be reduced and performance can be improved. Additionally, this network architecture allows for transfer across environment size changes, in addition to the type of transferred facilitated by the state-action space design.

1.3.2 Auxiliary Task Learning with Multiple State Representations

We specifically look at learning final policies that work with only the sensors available on the robot at test time. This may include things like cameras, range finders and internal joint sensors. The downside of learning a policy with high-dimensional states such as images is the increase in training time and required training data. However, we can reduce the penalty of using such a representation if the agent simultaneously learns a policy with a more task-specific, "feature-based" state representation. This feature-based representation may require additional sensors and preprocessing that would make running the feature-based policy outside the lab infeasible. However, these types of sensors are often required to extract a reward signal in the real world, so it makes sense to utilize this additional information during training time. Learning a feature-based policy is often faster, which can allow for better data collection and exploration, which in turn can speed-up learning with the image based representation.

The method in this thesis builds off of existing work in auxiliary task learning to create a method where tasks differ not only in rewards, but also in their state-spaces. Using such a technique, data collected while executing a task in the train-time, low-dimensional, feature-based state-space can be reused to speed-up training in the train-time, high-dimensional state-space. Additionally, this technique is compatible with other auxiliary task work which allows for using tasks with different rewards to guide exploration and speed-up the process of collecting useful samples for the network optimizers.

1.4 Robots and Tasks

In this section we give a brief overview of the two main robot task environments we use to test and demonstrate the thesis contributions. More details for these environments are given in the appropriate section. Additionally, the tasks here are not exhaustive, but only the main tasks used throughout multiple chapters and sections. We use both simulated and real-world versions of these robots and their associated tasks.

RoboCup Tasks RoboCup is an international competition where teams of researchers compete to create teams of autonomous robots that play versions of soccer. [101] Learning the whole game end-to-end is outside the scope of this thesis. However, we use various sub-tasks both single and multi-agent from this domain to test our contributed techniques.

RoboCup Small Size League (SSL) Robots For the RoboCup tasks, we use RoboCup Small Size League (SSL) robots, specifically the CMDragons robots. [17, 64, 104, 110] These are small, omni-wheel robots which are velocity controlled via radio commands. The robots have the capability of basic ball manipulation via contact as well as flat kicks across the ground and chip kicks which arc the ball through the air. We test our techniques both with simulated and real versions of these robots.

Ball-in-a-Cup Task The Ball-in-a-Cup game consists of a handled cup, with a ball hanging under the cup via a string attached to the handle. [88] The goal is for the player to swing the cup and catch the ball inside of the cup. Figure 1.1 shows an illustration of this task. A successful execution of the task typically consists of three phases: start, swing-up, and catch. At the start the ball hangs below the cup with no motion. During the swing-up phase the arm holding the cup swings the cup back and forth to build up momentum in the ball. After the ball has enough momentum to swing up past the top of the cup, the catch phase starts. The arm must position the cup to catch the falling ball and complete the task.

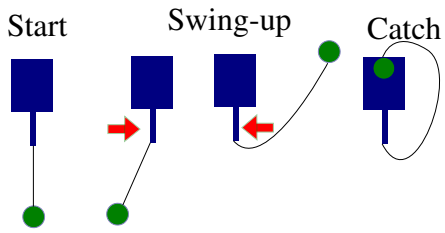


Figure 1.1: Illustration of the Ball-in-a-Cup task.

Sawyer Robot For the Ball-in-a-Cup task, we utilize both a simulated and real Sawyer robot from Rethink Robotics. [1] This is a 7 Degrees of Freedom (DoF) robot arm with various modes of joint and tool space control. For our tasks we utilize joint velocities. The robot has built in compliance through the series elastic actuators, which can help the robot safely learn policies in the real world at the expense of some control precision.

1.5 Simulation vs. Reality

Throughout this thesis we focus on techniques that work for real robots vs techniques that solely work in simulation. Simulations have a reality gap due to differences in physical modeling. Additionally, many simulations, unlike the real-world, act completely synchronously with the policy decisions, meaning no changes in the environment happen while the agent is deciding the next action and there is no real-time constraint on the agents decision making.

Despite the differences between simulation and reality, we make heavy use of simulation throughout this thesis. Simulations are still useful for running large scale experiments and collecting large amounts of data under controlled circumstances which can help highlight differences between techniques under ideal conditions. Additionally, if the reality gap is taken into

[1] `http://mfg.rethinkrobotics.com/intera/Sawyer_Hardware`

account when developing the techniques, it is possible to train in simulation and execute on the real-world hardware. Differences such as physical modeling errors can sometimes be lessened via appropriate choices of the state-action space and simulations can be designed to act asynchronously. We consider a technique appropriate for real-world robots if at a minimum trained policies can be executed on real hardware. However, the ideal case is that training also works directly on the real-hardware so that these simulation reality gap concerns can be ignored.

1.6 Contributions

This thesis makes the following robot deep reinforcement learning contributions:

- A new representation using multi-dimensional tensors for both states and actions that allows for zero-shot transfer across number of robots and objects in an environments.
  - A general description of how to map from the original task state-action spaces to this designed tensor state-action space
  - A specific mapping that works well for environments where robot and object positioning are important for task performance
- A policy network representation that works in tandem with the tensor state-action space to allow transfer across environment sizes.
- A method for using different state representations during training time to produce a flexible policy at test time using only the available robot sensors
  - Demonstration that such a technique can speed-up training on high-dimensional states, such as images
  - Demonstration that such a method works on both simulated and real-robots
  - Demonstration that such a technique can learn a sparse reward policy directly on real robots
- Demonstration of the various approaches on dynamic robot tasks, both in simulation and on real-robot systems, including systems and tasks such as:
  - Ball-in-a-Cup with a robot arm
  - RoboCup Robot Soccer related domains

1.7 Reading Guide to the Thesis

The following outline summarizes each chapter of the thesis. Chapter 2 provides an in-depth look at the RoboCup SSL domain and how standard Deep RL can be applied to such a robot and domain. We also use this example to point out shortcomings with existing techniques that are addressed by the thesis contributions. However, if you are familiar with RoboCup, Deep RL applied to robots and the associated shortcomings, this chapter is not necessary for understanding the thesis contributions. Instead, refer directly to chapters 3 and 4 for the presentation of the thesis contributions.

Chapter 2 — Assessing Shortcomings of Current Reinforcement Learning for Robotics gives an overview of the main shortcomings that arise when applying existing Deep RL techniques to robotic systems. The RoboCup Small Size League (SSL) domain, which is used throughout this thesis, is introduced and used as an example to illustrate these shortcomings on a real robot system performing various tasks. Finally, the chapter presents a short summary of how we address these shortcomings through the thesis contributions. This chapter can be skipped if the reader is familiar with RoboCup and Deep RL.

Chapter 3 — Tensor State-Action Spaces as Transferrable Representations describes how multi-dimensional tensors can be used to represent both state and actions, such that zero-shot transfer is possible as the number of robots and other objects in an environment vary. Additionally, we introduce the Fully Convolutional Q-Network (FCQN) network architecture, which works with this representation to allow for zero-shot transfer across different environment sizes. We show how this technique works for environments where positioning is important using different domains inspired by RoboCup SSL. We also show the generality of this representation to other domains via transfer results from a modified version of Atari Breakout. Finally, we demonstrate that despite the representation being at a higher abstract level, the policies learned in simulation can be applied to real robots using real sensor data.

Chapter 4 — Auxiliary Task Learning with Multiple State Representations introduces the auxiliary task framework used in this thesis, including the contributed method for allowing learning simultaneously with different state representations. This method allows for the state to include additional information at test time, that may come from additional sensors only available in a lab environment, and then at test time use only the sensors on the robot and in the test environment. For high-dimensional test-time states, such as those containing camera images, this method can also significantly speed-up training. We discuss how auxiliary tasks can be designed and demonstrate how they can apply to simulated RoboCup SSL tasks. Finally, we more fully introduce the Ball-in-a-Cup with a robot-arm domain and show that the thesis approach is capable of learning a policy from pixels with sparse rewards directly on the real-robot hardware.

Chapter 5 — Related Work reviews the literature related to the approaches presented in this thesis.

Chapter 6 — Conclusion and Future Work concludes the thesis with a summary of its contributes, and presents potential directions for future related research.

Chapter 2

Assessing Shortcomings of Current Reinforcement Learning for Robotics

This chapter points out shortcomings of existing Deep Reinforcement Learning (RL) techniques when applied to robots. We begin by discussing what we view as some of the largest challenges in applying existing Deep RL to robots. We demonstrate these shortcomings by applying RL to tasks related to RoboCup Small Size League (SSL) soccer. This domain is used throughout the thesis and serves as a good introduction to the types of problems the thesis contributions address. After showing how current technique apply to such a domain, and how the identified shortcomings manifest themselves, we discuss how the thesis contributions address these shortcomings.

2.1 Shortcomings of Existing Deep RL Approaches

While it is possible to apply existing Deep RL algorithms to robots, there are number of shortcomings that prevent wide-scale application. Three of the biggest shortcomings are:

1. Large training time and number of samples required
2. Adaptability of the policy to changes such as environmental changes, or number of agents/objects in the environment
3. Safety while training and executing

Each of these issues are discussed in more details in the following sections.

2.1.1 Sample Complexity and Training Time

The first major issue is sample complexity and training time. When training with complex robots, dynamic tasks, and high dimensional sensor data the number of training samples can be quite large. Especially, when training in the real world, we want to minimize the amount of training time and data, as unlike simulation, real-world training cannot run faster than real-time. At best, one can parallelize the learning using multiple robots and environments, but this can be prohibitively expensive depending on the robot and tasks. Additionally, the less time spent training before reasonable behavior is learned, the less chance there is for the robot to violate safety

constraints and damage itself or the environment.

Another aspect affecting sample complexity is the reward function used. Ideally, we would train with a sparse reward function, as this most accurately captures the goals of a task. However, given random exploration, in a large state-action space, it is unlikely for a robot to hit these sparse reward state-action pairs, and therefore, the amount of training data required can be unmanageable for many useful robotic tasks.

2.1.2 Adaptability

The second shortcoming, is the trained policies lack of adaptability to changes in the environment. The more adaptable trained policies are, the less the training time and sample complexity matters. If a policy is applicable for only a narrow set of task configurations and environments, then to deploy a RL approach for a real-world robotic task will require training a large number of policies to handle the different variations. Combined with the potentially large cost for training, this quickly becomes infeasible for many useful, real-world robotic tasks. Alternatively, if policies are adaptable to changes in the environment, then even a large amount of up-front training can be amortized across a range of similar environments. Similarly, if the policy is generalizable to changes in the environment, then training can happen in a more restricted lab setting or simulation environment and still apply to final robotic system.

2.1.3 Safety

Finally, keeping the robots and the robots' environment safe during training and execution presents a major obstacle. Given the randomness of exploration and an uninitialized policy, it is very likely that a robot will attempt dangerous actions. These can lead to robot breakages or damages of the surrounding environment. One approach to mitigate safety concerns is to train in simulation and then apply the learned policies to real robots via sim2real techniques. [3, 72, 83, 84, 106]. However, there is always a mismatch between simulations and real-systems, and a big enough gap will lead to policies that work in simulation but not in the real-world. Other approaches use standard RL techniques, but engineer safeties such that the robots are confined to a safe workspace, with only safe actions available at any time [72, 89]. While these approaches work, it can be significant effort to setup these systems and may include the need for additional sensors and equipment which make it difficult to train or test policies outside of a lab environment. More exotic techniques focus on optimizing the rewards for the RL agent such that unsafe actions are avoided while still allowing the original policies to execute. [4, 77] However, these techniques are still under-developed for robotic systems.

Even after training is finished, safety is still a concern. Unlike classical robotic algorithms, the policies do not have many guarantees about behavior. Instead, behavior is empirically evaluated along metrics such as reward per episode and success rate. Given that the policies lack guarantees regarding robustness and safety, it is common to still run the trained policies with hard-coded safeties to prevent clearly unreasonable action choices. Therefore, even if simulation is used to avoid safety issues during training, one still has to worry about safety at test time when executing on the real hardware.

2.2 Applying RL to RoboCup Domains

This section now demonstrates an application of existing RL techniques to the RoboCup SSL domain. We begin by introducing the RoboCup domain along with a description of the robots used and the tasks learned. We show, through this domain that while it is possible to do Deep RL with robot systems, the shortcomings identified in the previous section come into play repeatedly.

2.2.1 RoboCup Description

RoboCup robot soccer is an international competition where researchers compete to create teams of autonomous soccer playing robots [101, 110]. There are many different leagues, each with different research goals. There is a the 2D simulation league which simulates circular omni-wheeled robots with partial observability [105]. There is a 3D simulation league featuring humanoid robots each with their own observations [105] There are also numerous real robot leagues, including those that use humanoid robots such as the Nao robots [103]. Middle Size League (MSL) features teams of decentralized wheeled robots which play with a full sized soccer ball [102]. Finally, the domain of focus in this thesis, the Small Size League (SSL), which features centrally controlled teams of omni-wheeled robots which play with a golf ball [104].

Small Size League (SSL)

In this thesis, we focus on the Small Size League (SSL) domain. Figure 2.1 shows an illustration of the SSL field setup. A full game of SSL soccer uses 11 vs 11 robots. Each robot has a unique computer vision pattern that identifies the robot, the robot's team, the robot's position and the robot's orientation. The ball uses a special color that can easily be detected and tracked. Overhead cameras send images to a central vision server, which detects the robot patterns and the ball(s) and sends the detection information to each team at approximately 60Hz. [15] Typically, teams use a single computer that receives this information, plans and computes new robot commands and then sends these commands via radio to the entire team. The whole process repeats at 60Hz.

Each team designs and builds their own robots. These robots can be of any design (with some limitations on ball manipulation strategies, volume, etc.). Most teams have converged on a close to cylindrical robot with 4 omni-wheels around the base. This thesis makes heavy use both simulated and real versions of the CMDragons' robots. The CMDragons have a long history of research and competition in RoboCup SSL having won multiple competition years. [17, 64] This thesis utilizes the robots as well as existing code and infrastructure developed by multiple team members over the years. Chapter 5 contains more information the research contributions from the CMDragons.

Figure 2.2 shows a picture of the CMDragons robot with and without the cover and next to a standard SSL ball. The robot is approximately 18cm in diameter and can move omni-directionally. In addition to navigation via the omni-wheels, the robots posses a dribbler bar, which can apply forward or backward spin to the ball so that the robot can "pull" the ball or push it forward when moving. The robots additionally have two means of kicking: flat kicks and chip

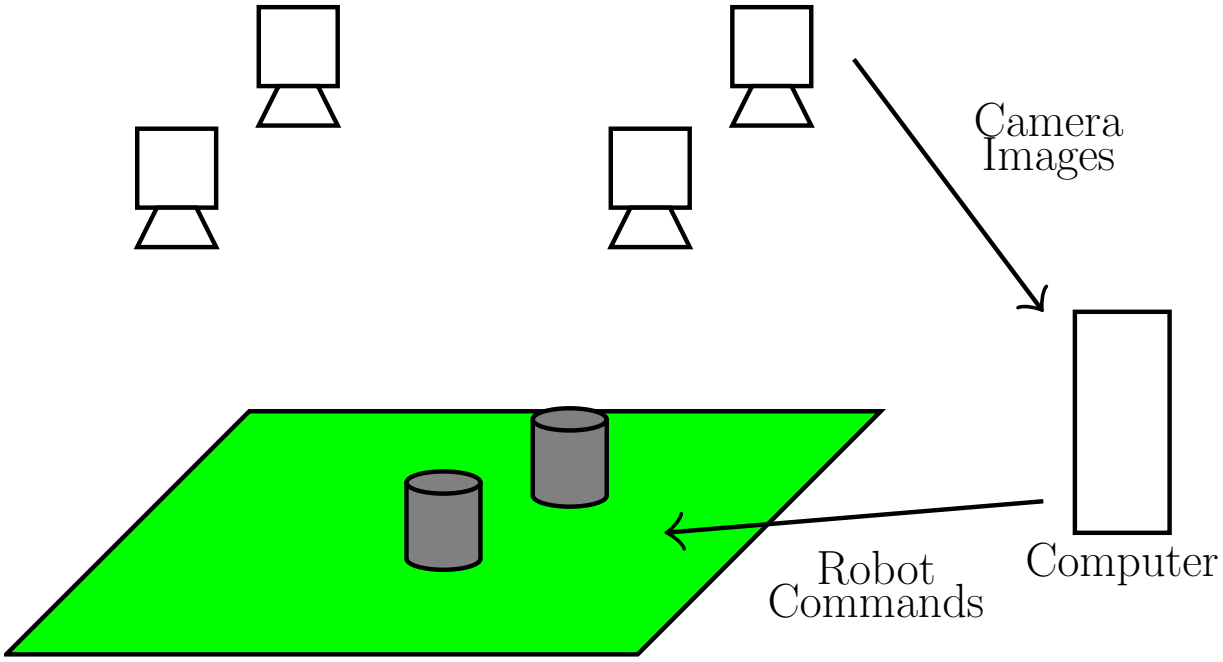


Figure 2.1: RoboCup Small Size League (SSL) field setup.

kicks. Flat kicks apply an impulse to the ball and keep the ball on the ground, whereas chip kicks launch the ball into the air in an arc.

Depending on the design, the robots can be controlled in different modes. For the CMDragons robots, each robot receives a specified wheel velocity for each wheel $[v_0, v_1, v_2, v_3]$. There is also a higher level egocentric velocity control $[v_x, v_y, \omega]$, which uses a model of the wheel configurations and the robot mass/inertia to translate to individual wheel velocities. [81].

Teams can use any onboard sensing and send back any information they desire. The CMDragons robots can send back limited information about the wheel states, battery levels and kicker state. The feedback rate is reduced as the number of robots being controlled increases. So for most tasks in this thesis we rely solely on feedback from the standard SSL vision system. [15]

2.2.2 SSL Tasks

Small Size League (SSL) presents many challenges to any RL approach. The environment is multi-agent, adversarial, uses real sensors and real robots, and contains many dynamic control tasks. Learning to play the entire game of robot soccer end-to-end would be an exciting project, however, it would be too large an undertaking for a single thesis. Instead, we use different pieces of the SSL robot soccer game from simple navigation control tasks to multi-agent team coordination games inspired by Keepaway [95, 96]. Tasks from the SSL league (and RoboCup

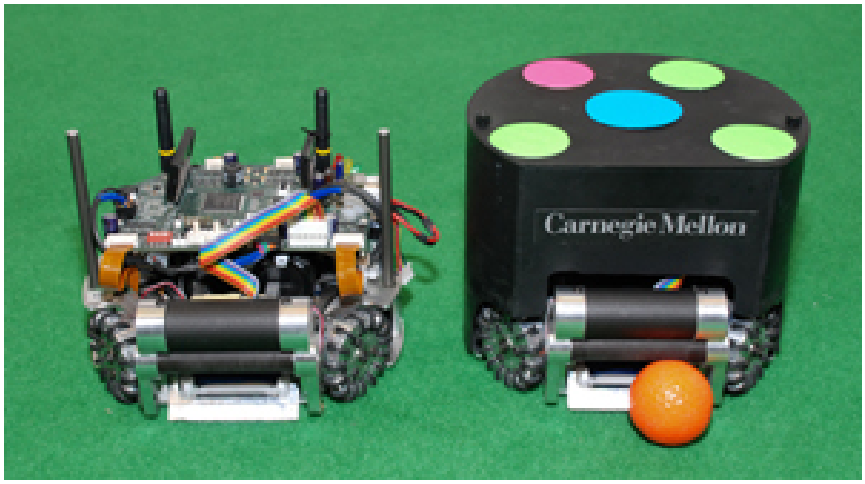


Figure 2.2: CMDragons robots with standard SSL ball.

in general) have been proposed as benchmark tasks for reinforcement learning [94, 96]. This has lead to a number of previous works investigating RL for RoboCup soccer [30, 78, 79, 96, 107] and a number of sub-task domains, some of which we use and adapt in this thesis. The remainder of this section gives a high-level overview of the tasks that are used in the experiments throughout this thesis.

Navigation

One of the most basic skills in RoboCup soccer is the ability to use the sensors on and off the robot to control the robot's motion on the field in order to position the robot for plays and navigate around opponents and obstacles. Depending on the speed of the robots, this can be a highly dynamic control task, where large wheel velocities can cause slippage or in the worst case a flipped over robot.

Ball Manipulation Skills

Another fundamental skill in RoboCup soccer is ball-manipulation. These include dribbling (moving the ball in a controlled manner through physical contact), passing between robots and shooting goals. Most of these skills require a fast control rate due to the dynamic nature of the task. Many forms of ball-manipulation skills have been previously studied in an RL context. Riedmiller et al. [79] used RL on real robots for the Middle Size League in order to learn ball manipulation skills. Recent Deep RL work from Hausknecht et al. [43] was able to learn, in simulation, to navigate to a ball and score on an empty goal using imitation learning. Later this was refined to learn, in simulation, from scratch [42].

Keepaway

Keepaway is a subdomain inspired by RoboCup robot soccer. It is multi-agent and can be partially observable (with each agent limited to its own sensor view of the environment) [95, 96].

The game consists of two teams of agents. One team starts with the ball, and attempts to maintain control of the ball as long as possible by dribbling and passing to teammates. An opponent team of agents attempts to intercept and take the ball from the opposing team as quickly as possible.

2.2.3 Problem Description

The existing CMDragons' team code used a Skills, Tactics, and Plays (STP) architecture [13]. STP is a hierarchical architecture consisting of three levels. Skills are coded policies that represent low-level tasks, used repeatedly in the game of soccer. These include: dribbling the ball, navigating to a point, etc. Tactics combine skills into behaviors for a single robot. Typically coded as state machines, where specific skills are called in each state. Plays are how multiple tactics are coordinated. Each robot is assigned a tactic based on cost functions, and then the robots execute these tactics independently.

Previously, all of the code for each level in this hierarchy is hand-written using classical robotics and planning algorithms. Low-level policies used algorithms such as Rapidly-exploring Random Tree (RRT) [14, 16, 56], while the tactics have been written using intuition and improvement through expensive testing. Ideally, we would be able to replace the entire STP hierarchy with a fully end-to-end, multi-agent policy. However, the amount of training data, time, and computation would likely be quite large. Also, there would likely be issues with choosing an appropriate reward function (ideally sparse), state representation (how to deal with robots as they enter/leave the game), action representation (how to deal with breakages, changing action space), and safety while training (e.g. do not crash while exploring new policies and break the robots). Instead, we will focus on a much smaller task: learning individual skills in simulation and then combine these skills manually with a simple state-machine into a tactic. These skill policies and the tactic can then be executed on the real robot.

Specifically, we learn three different skills from scratch in simulation: `go-to-ball`, `turn-and-shoot`, and `shoot-goalie`. Each skill is trained from scratch in simulation to avoid issues with long training times and safety. After each skill has converged we can chain them together to navigate the robot to the ball and score on an empty goal in simulation. Finally, we can take these trained skills and test them directly on the real robot hardware.

2.2.4 Deep Reinforcement Learning of Soccer Skills

This section introduces the basic notation used for our reinforcement learning problems. We also give an overview of the algorithms used.

Markov Decision Process (MDP)

Markov Decision Processes (MDPs) are a mathematical representation of a class of sequential decision making problems [75]. An MDP is defined by a tuple (S, A, R, T, γ). There is a set of states S and a set of actions A available to the agent. T is a function $p(s_{t+1}|s_t, a_t)$ which describes the probability of transitioning from state s_t to state s_{t+1} under action a_t. R is a reward function $R : (s_t, a_t) \to \mathbb{R}$. $\gamma \in [0, 1)$ is a discount factor which trades off short term rewards for long-term rewards. To solve an MDP, a policy $\pi : s \to a$ must be found such that the cumulative

discounted reward $\mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) | a_t \sim \pi(s_t), s_{t+1} \sim p(\cdot|s_t, a_t), s_0 \sim p(s)\right]$, where $p(s)$ is the initial state distribution. Reinforcement Learning is a method of solving and MDP when the transition function T and reward function R are not known ahead of time. Instead an agent must try actions in different states in order to gather data about the transition function and reward function in order to find the optimal policy for the MDP.

Reinforcement Learning

Model-free RL, which we use throughout this thesis, determines a policy by learning a state-action value function, commonly called a Q-function. [115] A Q-function is defined for a policy as $Q^\pi(s_t, a_t) = R(s_t, a_t) + \mathbb{E}_\pi \left[Q(s_{t+1}, \pi(s_{t+1}))|s_{t+1} \sim p(\cdot|s_t, a_t)\right]$. Which represents the cumulative reward of choosing action a_t in state s_t and then following the policy thereafter. Off-policy methods will follow some behavior policy to collect data while estimating the optimal Q-function, Q^*. Typically, this behavior policy is the current best policy estimate with some exploration actions added in.

Continuous action space policies generally cannot represent a policy as a Q-function, as choosing the optimal action would require a continuous optimization of the current Q-function to choose the best action. Instead, typically an actor-critic method is used. An actor which represents the policy mapping from a state to a continuous action is learned. A critic representing a Q-function evaluates the actions chosen by the actor in order to update the actor's policy. Once training is complete, only the actor function is needed. In our case, we will train three pairs of actor/critic networks, one for each skill. At test time we will execute the three actor networks, one for each skill.

We specifically train using the Deep Deterministic Policy Gradient (DDPG) algorithm. [60] DDPG is an actor-critic method suitable for learning in continuous states and action spaces. Like in most Deep RL algorithms, it utilizes a replay buffer, which stores (s, a, r, s') tuples experienced while training. To update the actor and critic networks samples are drawn from this buffer and combined into random mini-batches. To further speed-up training and reduce sample complexity we start the replay buffer with simple, non-optimal demonstrations of the skills as in Vecerik et al. [109]. We also utilize reward shaping to help the critic network converge faster.

2.2.5 Environment Setup

We assume that the standard SSL vision system is available when learning and testing the skill policies. This means a feature-based representation can be used, where features are derived from the position and orientation information provided by the vision system. All skills use coordinates relative to the robot's frame of reference. Figure 2.3 shows an illustration of the robots frame and the world frame of reference. (x^R, y^R) is the robot frame and (x^W, y^W) is the world frame of the environment.

The robot is commanded via velocities in the robots frame of reference at 60Hz. We also allow the robot to control the speed of the dribbler bar spin, which affects how much backspin is put on the ball.

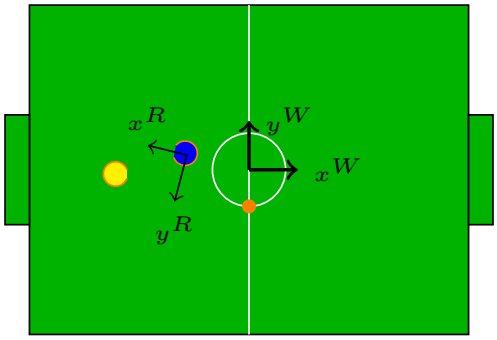


Figure 2.3: Robot and world frame definitions used to calculate features for each skill.

`go-to-ball` Skill

The goal is for the robot to end with the ball positioned on it's kicker and dribbler. The robot starts with some random position and orientation, while the ball starts with some random position on the field.

We use the following state features for this skill:

$$s = (P^B, V^R, \omega^R, d_{r-b}, P^R_{top}, P^R_{bottom}, P^R_{left}, P^R_{right}) \tag{2.1}$$

where P^B is the x-y location of the ball, V^R is the robot's linear velocity, ω^R is the robot's angular velocity, d_{r-b} is the distance from the robot to the ball and P^R_{top}, P^R_{bottom}, P^R_{left}, P^R_{right} are relative coordinates to the closest points on the top, bottom, left, and right edges of the field to the robot. Figure 2.4 shows an example of these features.

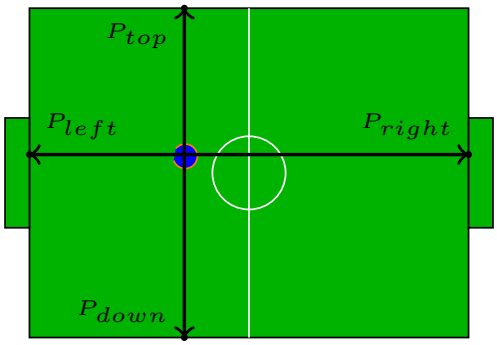


Figure 2.4: `go-to-ball` features.

The action space of the robot is: (v^R_x, v^R_y, ω^R) where v^R_x is the x-velocity of the robot, v^R_y is the y-velocity of the robot, and ω^R is the angular velocity of the robot.

The reward function is a combination of a reward for being in contact with the ball and a shaping term based on the relative distance and orientation of the robot's dribbler to the ball. The reward function is:

$$r_{total} = r_{contact} + r_{distance} \tag{2.2}$$

$$r_{contact} = \begin{cases} 100 & \text{ball on the dribbler} \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

$$r_{distance} = \frac{5}{\sqrt{2\pi}} \exp -\frac{d_{r-b}^2 + \theta_{r-b}^2}{2} - 2 \tag{2.4}$$

where θ_{r-b} is the angle between the forward direction of the robot and the ball.

We seed the replay memory with simple demonstrations of the robot driving straight into the ball. The robot is spawned facing the ball some distance away and commanded to drive straight for the whole episode.

`turn-and-shoot` Skill

The goal is to learn to dribble the ball in place so that the robot faces the goal, at which point the robot should shoot the ball at the goal. The robot must learn the proper control of dribbler bar speed and rotation velocity, as well as proper kick strength and timing. For this skill there is no goalie. The robot starts with some random position and orientation on the field. The ball starts on the robot dribbler.

The state features are:

$$s = (P^B, V^B, \omega^R, d_{r-g}, \sin(\theta_l), \cos(\theta_l), \sin(\theta_r), \sin(\theta_r), \mathbb{I}_{ballkicked}) \tag{2.5}$$

where P^B is the x-y location of the ball, V^B is the linear velocity of the ball, ω^R is the angular velocity of the robot, d_{r-g} is the distance from the robot to the goal, θ_l is the angle between the front of the robot and the left goal post, θ_r is the angle between the front of the robot and the right goal post, and $\mathbb{I}_{ballkicked}$ indicates if a kick command greater than 0 has been issued by the agent this episode. Figure 2.5 shows an illustration of how these features are calculated.

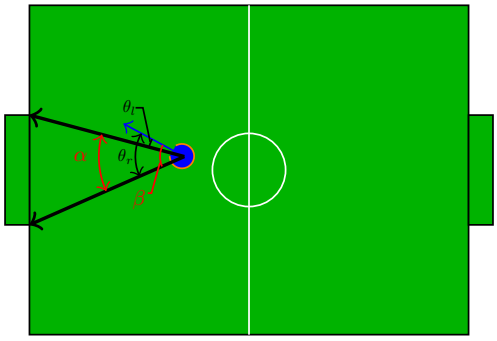


Figure 2.5: `turn-and-shoot` features.

The action space of the robot is: $(\omega^R, dribble, kick)$ where ω^R is the commanded angular velocity of the robot, dribble is the dribble motor speed, and kick is the command kick power.

The reward function is shaped so that fast kicks when the robot is faced between the goal posts give high reward, fast kicks when not aimed at the goal result in high negative rewards, and losing the ball while dribbling provides negative reward. The reward function is:

$$r = \begin{cases} \exp \frac{(\alpha-\beta-0.05)*|V^B|}{5} - 1 & \text{ball kicked} \\ -0.5 & \text{ball not on dribbler} \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

where α is the angle between the goal posts relative to the robot's position, and β is the larger one of the angle differences between the kicking direction and the two goal posts.

We seed the replay memory with demonstrations by spawning the robot facing towards the center of the goal and sending a high kick command. These demonstrations give some initial training data with positive rewards, without demonstrating any of the dribbling and aiming parts of the skill.

`shoot-goalie` Skill

This skill is like the `turn-and-shoot` skill, but this time there is a static goalie. This makes the aiming portion of the skill more difficult.

The state space is the same as `turn-and-shoot` but with the following additional features: $\sin(\theta_c^{goalie})$, $\cos(\theta_c^{goalie})$, $\sin(\theta_l^{goalie})$, $\cos(\theta_l^{goalie})$, $\sin(\theta_r^{goalie})$, $\cos(\theta_r^{goalie})$. Where θ_c^{goalie} is the angle of the front of the robot to the center of the goalie, θ_r^{goalie} is the angle of the front of the robot to the right side of the goalie, and θ_l^{goalie} is the angle of the front of the robot to the left side of the goalie. Figure 2.6 shows an illustration of these additional state features.

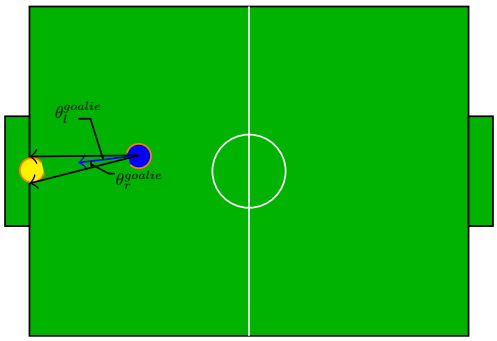


Figure 2.6: `shoot-goalie` additional features. The other features are the same as `turn-and-shoot` skill features shown in figure 2.5.

For this skill, we use a constant dribbler speed, and let the robot only control angular velocity and the kick strength: $(\omega^R, kick)$.

We utilized a sparse reward function based on successfully or unsuccessfully completing an episode. The reward function is:

$$r = \begin{cases} 1 & \text{episode success} \\ -1 & \text{episode not successful} \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

An episode is successful when the ball is in the goal after crossing the goal line at least 100mm away from the side of the goalie. The restriction on distance to the goalie prevents the robot from learning a strategy that ricochets the ball of the goalie robot into the goal. In this case, we do not seed the replay memory with any kind of demonstration.

2.2.6 Empirical Results

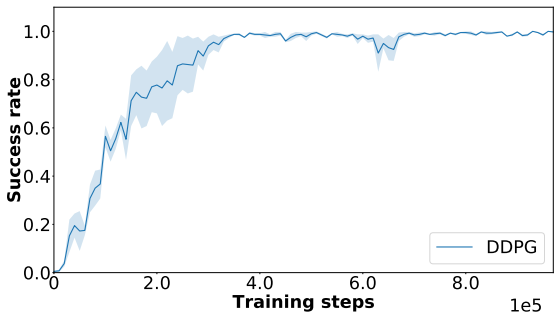
This section presents results of the simulation training, including final policy performance as well as learning curves from throughout training. We also present the results of the policy performance when applied, with no further training, to real robot hardware.

Training Parameters

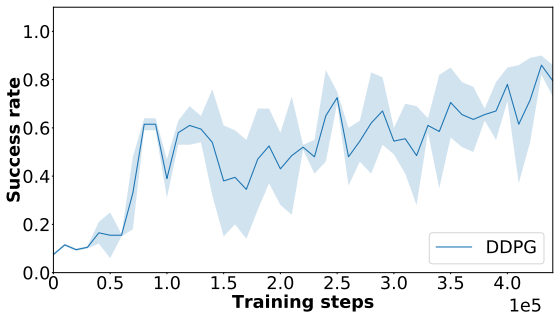
We use the same network structure for each skill, adjusting only the replay memory size and noise parameters. Both the actor and critic are two fully connected layers with ReLU activation. Actions from the actor enter the critic as inputs to the second layer. Appendix A.2 gives more details as to specific layer sizes, learning rates, etc. For the `go-to-ball` and `turn-and-shoot` skills we initialize the replay memory with 10,000 demonstration transitions.

Simulation Results

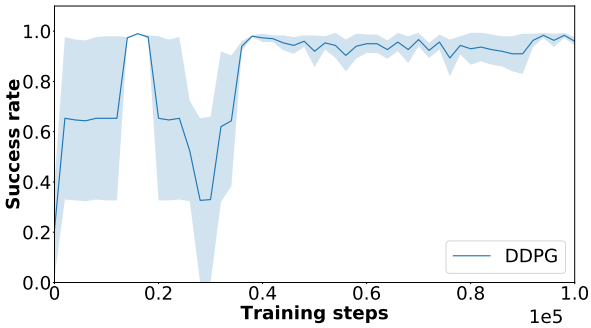
Figures 2.7a, 2.7b, and 2.7c show the training curves for the three skills. To generate this curve, network weights throughout training were saved. Each set of saved weights was then run, with no noise 100 times and the results were averaged. This whole process was repeated from scratch 3 times and average to produce the mean curves shown and the shaded regions depicting the variance between the different independent runs. All three skills were successfully learned, with the `go-to-ball` skill converging fastest and being the most stable.



(a) `go-to-ball` skill



(b) `turn-and-shoot` skill



(c) `shoot-goalie` skill

Figure 2.7: Training curves for all three skills.

We compared the learned skill performance against the hand-coded version of the skills. Table 2.1 shows the results of this comparison. For each skill we ran 1000 episodes of the trained skill and 1000 episodes of the hand-coded skill and tracked the success rate. The original hand-coded `turn-and-shoot` skill optimized for time taken for a shot. For this reason, the robot would take risky shots near the edge of the goal, which lead to a reasonably high failure rate. In the absence of a goalie, optimizing for speed isn't necessary. So we adjusted the hand-coded skill to optimize for accuracy and re-ran the comparison.

	Success Rate	
	Hand-coded Policy	**Trained Policy**
`go-to-ball` skill	**1.0**	0.999
`turn-and-shoot` skill	0.71	**0.878**
`turn-and-shoot` skill (fine-tuned)	**0.948**	0.878
`shoot-goalie` skill	0.978	**0.99**

Table 2.1: Comparison of hand-coded policy and trained policy

We also compared the average time taken between the trained and hand-coded policies. Table 2.2 shows the result of this comparison. The trained `go-to-ball` skill takes slightly longer than the hand-coded `go-to-ball` skill. Qualitatively, the learned skill looks more "organic",

the robot makes a smooth curve towards the ball, which is not necessarily optimal. Whereas the hand-coded skill does a linear interpolation between the starting position and orientation and the desired position and orientation, which leads to a straight line path that can take less time. The other two skills are much closer in terms of time taken to completion.

	Hand-coded Policy (s)	**Trained Policy (s)**
`go-to-ball` skill	$\mathbf{1.46 \pm 0.47}$	2.11 ± 1.19
`turn-and-shoot` skill	$\mathbf{2.11 \pm 0.99}$	2.35 ± 2.01
`shoot-goalie` skill	1.59 ± 0.36	$\mathbf{1.56 \pm 0.76}$

Table 2.2: Comparison of time taken between hand-coded policy and trained policy

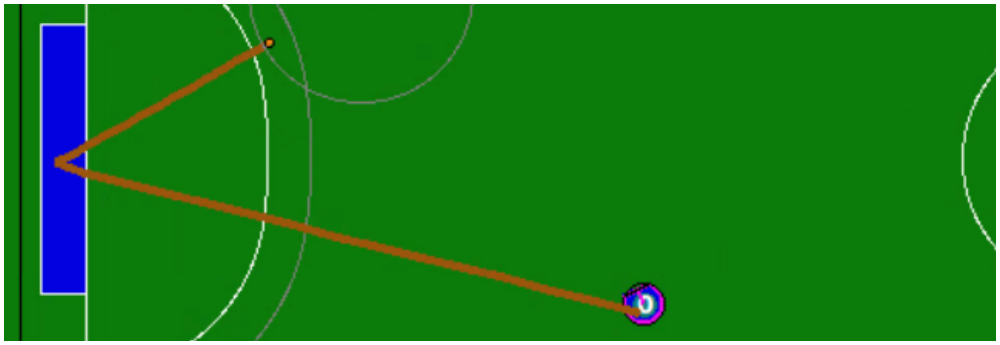
Figure 2.8 shows a series of images from the execution of the `go-to-ball` skill followed by the `turn-and-shoot` skill in simulation. The blue circle shows the robot. The blue line shows a short history of the robot's trajectory. The orange circle is the ball. The orange line shows a short history of the ball's trajectory. The robot begins executing the `go-to-ball` skill in figure 2.8a. The robot heads towards the ball adjusting its heading along the way. The robot gets the ball on its dribbler in figure 2.8b, which finishes the `go-to-ball` skill. The robot begins executing the `turn-and-shoot` skill in figure 2.8c until it is aimed at the goal as shown in figure 2.8d. Once aimed the `turn-and-shoot` skill kicks the ball and scores a goal as shown in figure 2.8e.

(a)

(b)

(c)

(d)



(e)

Figure 2.8: Images from execution of `go-to-ball` and `turn-and-shoot` skills on simulated SSL robot.

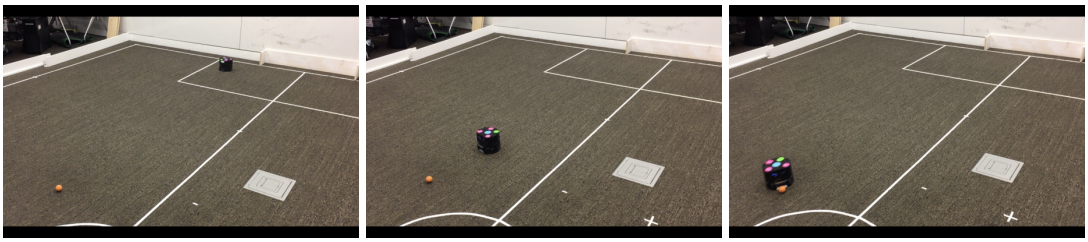
Real Robot Results

After training and testing the policies in simulation, we ran the final learned policies on the real robot hardware. No additional training was performed on the real-robot hardware. The robot and ball were put in different configurations for each episode. Table 2.3 shows the results of our evaluation. We tested 7 runs for `go-to-ball`, 36 runs for `turn-and-shoot` and 12 runs for `shoot-goalie`. `go-to-ball` succeeded 6 out of 7 times, `turn-and-shoot` succeed 33 out of 36 runs and `shoot-goalie` succeeded 11 out of 12 runs. Figures 2.9, 2.10, 2.11 show images from the real robot tests.

Skill	**Number of Successes**	**Number of Trials**	**Success Rate (%)**
`go-to-ball` skill	6	7	85.7
`turn-and-shoot` skill	11	12	91.7
`shoot-goalie` skill	33	36	91.7

Table 2.3: Success-rate of simulation skill trained policies on real-robot hardware

Figure 2.9 shows a sequence of images from the execution of the `go-to-ball` skill. Figure 2.9a shows the robot and ball at the initial position when executing a trial of the `go-to-ball` skill. Figure 2.9b shows the robot approaching the ball and turning towards it. Figure 2.9c shows the robot successfully achieving the goal, ending with the ball on its dribbler.



(a) (b) (c)

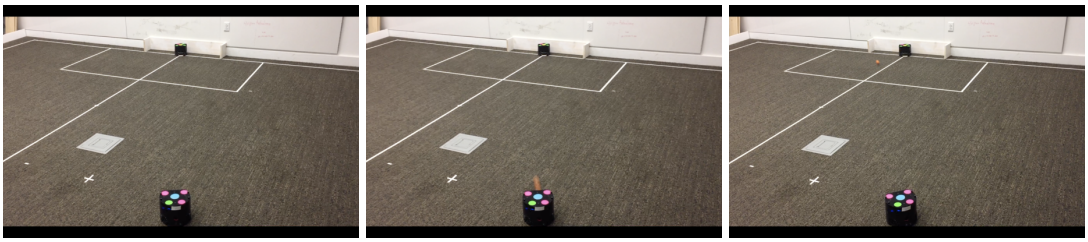
Figure 2.9: `go-to-ball` skill on real robot

Figure 2.10 shows a sequence of images taken from the execution of the `turn-and-shoot` skill on the real robot. Figure 2.10a shows the initial configuration of a trial of the `turn-and-shoot` skill on the real robot. The robot starts with the ball touching the dribbler. Figure 2.10b shows the robot beginning to turn while dribbling the ball. Finally, figure 2.10c shows the ball after being kicked as it is heading towards the goal.

(a) (b) (c)

Figure 2.10: `turn-and-shoot` skill on real robot

Figure 2.11 shows a sequence of images taken from the execution of the `shoot-goalie` skill on the real robot. Figure 2.11a shows the initial configuration of the `shoot-goalie` skill executing on the real robot. Figure 2.11b shows the robot kicking the ball towards the gap between the goalie and edge of the goal. Figure 2.11c shows the ball as it is almost entering the goal following the kick in figure 2.11b.



(a) (b) (c)

Figure 2.11: `shoot-goalie` skill on real robot

2.2.7 Discussion of Shortcomings in the RoboCup Domain

We now analyze how the shortcomings identified at the start of this chapter manifested themselves in the RoboCup domain. We discuss how such issues were engineered around and why further improvements to these areas are needed in order to make Deep RL more widely applicable to robots.

Training Time and Data

We mitigated the effect of training time and data in this domain through three techniques:

1. Running the simulation faster than real-time
2. Seeding the replay memory with non-optimal, but correct demonstrations of simple cases
3. Heavy reward shaping

The first approach is only viable when training in simulation, but in this thesis some of the techniques are applied directly on real-robot hardware. Additionally, it is not always trivial to

create a realistic model of a task. Tasks that are highly dynamic, have many contacts, use high-dimensional sensors or have difficult to simulate objects (such as deformable objects like ropes) may be difficult or impossible to accurately simulate.

Seeding the replay memory and shaping the reward gave the learner useful data early on to quickly learn reasonable behavior that it could refine through further exploration. [109] The danger of such a technique is that it biases the policies towards particular solutions to a problem. Depending on the strength of the bias, the learner may converge to a local optimum and never discover a better approach. Additionally, it is not always possible to come up with a non-optimal policy for a task up-front. At which point, this technique cannot be used.

Finally, we made use of heavy reward shaping, taking advantage of our expert knowledge of the tasks. While such shaping can help in some tasks, changing the reward function biases the agent towards particular behaviors. Depending on how the reward is changed, it is very easy to make a policy that would be optimal in the sparse reward case, non-optimal. Or worse, make a policy that does not accomplish the task at all, the optimal policy for the shaped reward.

2.2.8 Adaptability of Policies

In our application of Deep RL to RoboCup soccer, we made no attempt to transfer policies to different environments other than the simulation to real transfer. And the simulation was designed to mimic the real world as closely as possible to minimize this gap. However, in real-robot soccer games the environment would vary a lot from our training and test environment. Specifically the environment would be sized differently, with the robot playing on a larger field with potentially different ratio of field height and width. Even more different, there would be multiple robots on the field, some cooperative and some adversarial. Our feature based state while compact and well designed for this particular task, would need to be changed to accommodate different numbers of robots. A change in the state representation would necessitate learning a new policy. These types of changes (environment size and varying number of agents and objects) are not unique to the RoboCup domains, but can come up with many robotic systems.

2.2.9 Safety while Training and Executing

Safety while training was mitigated by our use of simulation. In simulation the robot can try anything and there is no risk to the real-hardware an environment. The agent can simply be provided with a negative reward for actions deemed dangerous and allowed to train until it learns to avoid such behaviors. As discussed, it is not always possible, or desirable to train in simulation.

Safety while testing was addressed via human intervention (i.e. stopping the agent when it was doing something dangerous) and engineered safety controls that modified the agent actions to prevent dangerous actions. Appendix A.1 gives details about how these safeties were implemented.

2.3 How this Thesis Addresses the Shortcomings

Finally, we conclude this chapter by giving an overview of how this thesis works to address the shortcomings identified in this chapter. While the main focus of the thesis is on reducing training time and increasing flexibility of learned policies through representation choices, we also discuss how the thesis contributions can contribute incidentally to safety.

2.3.1 Improving Policy Adaptability

One of the main contributions of this thesis is designing a representation for state, actions and policies that allows for zero-shot transfer as the number of agents and objects in an environment vary and the size of the environment varies. The goal is to allow for a single policy to work across a wide variety of changes at test time, thus training cost, even if large, can be amortized across these different domains. The thesis approach designs a tensor state-action space and a special network architecture that uses only convolutions and deconvolutions. For 2D environments this ends-up being an image-like representation for tasks. Specifically we focus on environments and tasks where positioning is important to completion. Our main test domains focus on environments inspired by RoboCup soccer, but positioning is important in many other robot tasks such as restaurant robots [68] and indoor navigation robots [29, 111]. Chapter 3 explains the representation, how learning is performed with such a representation, and presents empirical results in both simulation and with simulation policies applied to the real SSL robots.

2.3.2 Reducing Required Training Time and Data

The second main contribution of this thesis is an extension of the Scheduled Auxiliary Controller (SAC-X) framework, such that tasks can vary not only by reward but in the representation of the state. [80] The goal is to leverage strengths of different state representations across tasks to shore up weaknesses in each particular representation. We show that this approach is capable of significantly reducing training time, especially for high-dimensional states such as camera images. Chapter 4 explains the auxiliary task framework, how auxiliary tasks are chosen, how learning with different state representations works and provides empirical evidence for simulated RoboCup SSL tasks and a real-world Ball-in-a-Cup performed with a robot arm.

2.3.3 Safety

While safety is always a concern when applying RL to real robot systems, this thesis assumes that in the short-term, it is possible to engineer safety into the system. Additionally, because of the lack of safety guarantees, even for trained policies, these types of engineered safety systems would most likely be required for any real-world deployment of an RL policy to a robotic system. Nevertheless, thesis techniques can incidentally improve safety.

Designing transferrable representations, means that rather than starting from scratch when the environment changes, the policy should maintain reasonable (but not necessarily optimal) behavior after transfer. This reasonable behavior should be far safer for the robot and the environment than random exploration with a brand new agent training from scratch. Additionally, as

we will show in our approach for tasks where positioning is important, our tensor state-action space can eliminate certain types of unsafe actions by making them not representable. Rather than allowing any random actions, which could for example crash the robots into walls, the agent actions can be restricted to valid position commands only.

Using the auxiliary task with different state representation technique we can significantly reduce training time. By reducing training time and samples we reduce the amount of time the robot has to make catastrophic mistakes before it assumes reasonable behavior. Additionally, our technique allows for different sensors at train and test time. This means we can use specialized sensor setups in the lab during training that make it easy to keep the robot safe during the most dangerous part of the learning phase, and then not require these setups at test time. This can make it easier to design the safety systems in use.

Chapter 3

Tensor State-Action Spaces as Transferrable Representations

As discussed in chapter 2, one of the shortcomings of existing Deep RL approaches is the lack of adaptability to changes in the environment. In this chapter, we present the thesis contribution which addresses this short-coming: tensor state-action spaces and the Fully Convolutional Q-Network (FCQN) architecture. [88, 90] By anticipating the types of changes we would like a policy to adapt to, before we begin training, namely changes in the environment size and changes in the number of agents and objects in the environment, we show that it is possible to design a representation that allows for zero-shot transfer across these changes. Specifically our approach uses multi-dimensional tensors for *both* states *and* actions. We show that this representation works especially well for tasks in which positional information is important to performance. We demonstrate our approach on simulated tasks inspired by RoboCup SSL. We show the generality of our approach by performing transfer in a modified Atari breakout game. Finally, we show that despite the abstracted nature of our tensor representation, the learned policies are capable of applying to real robot hardware and real sensor data.

3.1 Problem Description

For many robotics problems, training an initial policy can be expensive. Additionally, for many robotic tasks, we would like to have policies that can adapt to changes in the environment. Specifically, in the case of the RoboCup SSL domains presented in chapter 2 we can anticipate that the robot's policy should work on different size fields and with different numbers of other robots on the field. Using the standard representations for states, actions, and policy network, we would need to train different policies for every scenario, or have some way of adapting the policy to the new environment changes online. As an alternative, knowing the types of environment changes we expect, we can try to design a representation for the states, actions, and policies that will allow the policy to apply with no additional learning required. Allowing the policy to transfer without additional learning, means that even if the initial training is slow, overall we may end up needing to do less training, as we only need to train one policy that can work across a number of environments. For many robotics problems, training an initial policy can be expensive. In this

thesis we explicitly consider two types of changes: changes to the size of the environment, and changes to the number of agents and objects in the environment.

While not true in all robotic tasks, spatial positioning of a robot in relation to the environment and other agents and objects is often very important to successful task completion. For example, in the RoboCup SSL domains from chapter 2, agents must know where they are located on the field, where the ball is, where their teammates are and where the opponents are at. In this thesis, we give special consider to transferrable representations for tasks where spatial positioning is important to the successful execution of a policy.

3.2 Tensor Representations

In this section, we introduce the basic notation used throughout this thesis contribution. We explain why standard feature representations present difficulties for transfer. We then explain how we can translate the standard state and action representations back and forth between our tensor state and action spaces. The new tensor state-action space is designed to not have the transferrability issues of the original representation, while still allowing for optimal policies to be representable.

3.2.1 Background and Notation

We assume that an environment can be modeled as an Markov Decision Process (MDP) as described in section 2.2.4. We assume that within the MDP environment there are some set of entities X, which are either agents or some other objects of relevance to the MDP task. Each entity will be associated with some features $\forall x \in X, F^x$. These features contain information relevant to the entity that is needed for the task such as positions and velocities.

As in normal Deep RL, the goal will be for each agent to find a policy π_θ, where θ are the parameters of the policy, that optimizes the discounted sum of rewards. Typically, such a policy will include a number of Fully Connected (FC) layers. The number of parameters in an FC layer are directly related to the input and output size of the layer. Thus the number of parameters will be directly related to the number of features used as the state input to the policy network. Therefore, any changes to the feature representation, including adding new features for new agents or objects, will require finding a new set of θ parameters.

A simple way of representing the full state of the MDP is to create a vector concatenating all of the features observations: $S = [F^0, F^1, \ldots]$. This standard state representation can allow for normal Deep RL policy learning, however, this representation will not allow for easy transfer as agents and objects are added or removed from the environment. This is because adding or removing agents and objects will add or remove features, which will change the feature vector length. As mentioned, this will change the size of the FC layers which will require a new policy parameterization for the changed representation.

We will consider scenarios with both discrete actions $a_d \in A_d$ and parameterized actions [42] $a_p(y) \in A_p$ where y is generally some entity in the environment. The full action set is the combination of both types of actions: $A = A_d \cup A_p$. A typical discrete action may be "move left". A parameterized action may be "pass ball to y", where parameters y is another agent in

the environment. Naïvely, Parameterized actions can be treated as discrete actions by grounding all parameters, which makes it possible to use any standard discrete action Deep RL approach to learn a policy. If using the grounded actions for learning, the dimensionality will vary when new entities are added or removed from the environment, as new groundings will be added or removed. Like with the state vector, if using FC layers in the policy, changing the dimensionality of the actions will require a change in the policy parameters θ. Requiring new parameters to be found when the environment changes makes transferring a learned policy much more difficult.

The goal in this work is to develop a representation for states and actions that can encode the original state and action space while remaining a fixed dimensionality regardless of how many entities are added or removed from the environment. If such a representation is used, then when the number of entities change, new new parameters need to be found and the same policy can be applied with no changes. Note, that this policy after transfer may not be optimal. We can invent tasks where the reward function or transition function changes drastically as entities are added and removed from the environment. However, in many domains the basics will remain the same, even if adjustments will be needed at the high level. For example, in the RoboCup domain, the basics of soccer should not change regardless of how many agents are on the field. While the optimal policy for 1-vs-1 soccer can be different from 11-vs-11 soccer, the basic objective is still for the team to control the ball and score using basic skills of navigation, ball manipulation, shooting, etc.

3.2.2 Tensor-based States and Actions

We will design a new multi-dimensional tensor state S_t that has a fixed dimensionality regardless of the number of entities in the environment. To use this new representation, we will design a mapping from the original feature vector state S to this new tensor state S_T. Similarly, we will construct a new multi-dimensional tensor action A_t that has a fixed dimensionality regardless of the number of entities in the environment. To use this new action representation we will design a mapping from this tensor action space A_t to the original action space A. With the original state-action space we learned a policy $\pi : S \rightarrow A$. Any change in the number of entities changed the dimensionality of S and A which required a new set of policy parameters. Instead, we can learn a policy $\pi_t : S_t \rightarrow A_t$. Because S_t and A_t are designed to not change dimensionality as we vary the number of entities, the policy will not need to be reparameterized.

Let $\phi : S \rightarrow S_t$ be an injective mapping function that converts the original state vector to this new tensor state with fixed dimensionality regardless of number of entities in the environment. This means that every state in S has one and only one representation in S_t, however, there may be additional values of S_t that have no representation in S. For some environments, such a grid-world, some Atari games, or RoboCup SSL, an image of the environment will satisfy these requirements. An image is just a special case of an arbitrary dimensioned tensor.

Let $\psi : A_t \rightarrow A$ be a surjective mapping function that converts from the tensor action space A_t back to the original action space A. This means that every tensor action can be mapped back to the original action space, however, multiple of the tensor action representations may map to the same action. Again we could consider an image like representation for the actions for domains such as Atari, RoboCup SSL, or grid-world environments. The key is that there must be some way of converting this image like representation back to the original action space. And in the

more general case the action will be a multi-dimensional tensor which may have more than the standard 3 dimensions of an image.

Given these mapping functions, we can learn a policy $\pi_t : S_t \rightarrow A_t$ instead of the policy $\pi : S \rightarrow A$. Figure 3.1 shows how this tensor based representation relates to the original MDP for a given example of a 5x5 grid world with 2 entities: agent 0 and object 0. Part (a) shows the feature based state, where each entity has 2 features. Part (b) shows the tensor based state created by applying the position mapping ϕ to the feature state in (a). Part (c) gives an example of the Q-value output of a policy network. Part (d) shows how the Q-values from part (c) are transformed into one-hot action tensors via the argmax function. Finally (e) shows the result of applying the action mapping function ψ to the one-hot action tensor output in order to produce an action in the original MDP action space. In this case this results in a_p with parameter object 0 being chosen as the action. A traditional MDP policy maps directly from (a) to (e) in the figure. Our approach takes the path (a) - (b) - (c) - (d) - (e) in the figure.

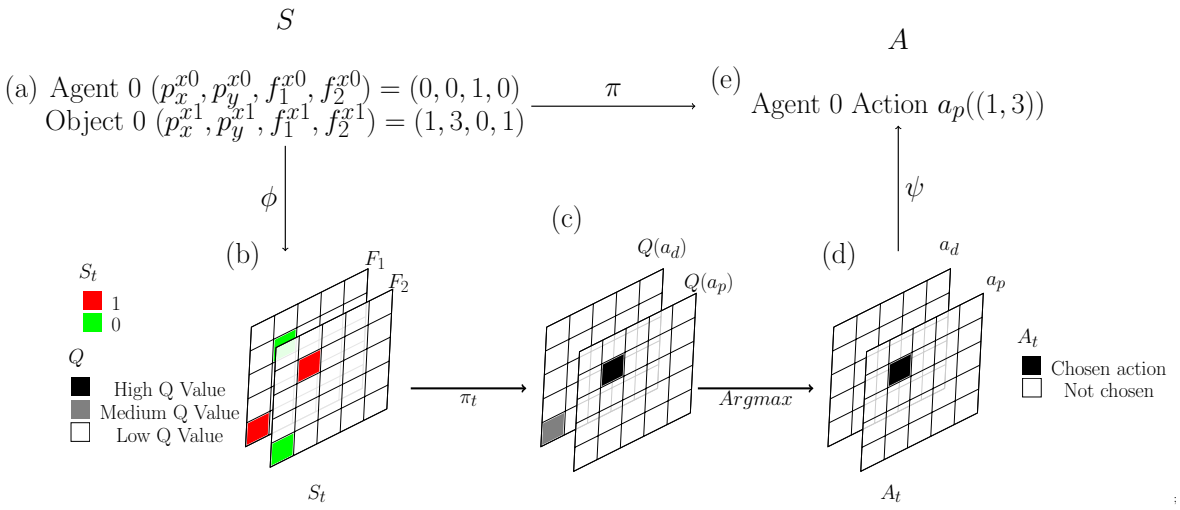


Figure 3.1: Example transformation to the tensor-based state and action representation.

3.2.3 Optimality of Policies with Tensor Representation

Because ϕ is injective, the new tensor state space is fully observable and can represent all states possible in the original state space. The ψ is surjective, meaning that for a given state, there is at least one action in the new action space that maps back to each possible action in the original space. So the policy may always select the same actions it would have in the original action space. Given these two properties, the optimal policy can be represented exactly in the new state action space. The benefit from this new tensor state-action space is that we can design it to be a fixed dimensionality for a given environment size, which allows a policy to apply with no new parameters needed as agents and other objects are added and removed from the environment.

3.3 Position Based Mappings

In the rest of this section, we consider tasks where all entities have position features. Tasks such as RoboCup SSL provide these features via a vision system. Other robotic systems may extract these coordinates via motion capture systems, SLAM algorithms, object detectors or a combination of all of these. Under the assumption that position information is available for all relevant agents and objects in the environment we give a specific pair of mapping functions ϕ and ψ.

3.3.1 Environments

Previously, we have mentioned the agent's "environment". Here we will formalize what is meant by this term in order to define the position based mappings. For tasks with heavy importance on spatial positioning the concept of the environment is usually implicitly encoded in the MDP transition function. For example, no transitions to positions outside of the environment are possible and the environment uses Euclidean space. An environment E can be defined for an MDP where the environment has some spatial dimensions D, where $E \in \mathbb{R}^D$. All entities must exit inside the environment bounds. To be observable, entities must have positions in the environment: $p^{x_i} = (p_1^{x_i}, \ldots, p_D^{x_i}) \in E$. If we assume finite environments, then $p_i \in [e_i^{min}, e_i^{max}]$ where e_i^{min} is the minimum value of dimension i of the environment and e_i^{max} is the maximum value of the dimension i of the environment. In the rest of the formulation, we assume that positions are either discrete or can be discretized to an appropriate level where the learning problem is still tractable. However, any feature other than the position coordinates may remain continuous.

3.3.2 Position based ϕ and ψ Mappings

If we assume each $x_i \in X$ has a position and a single feature f^{x_i} representing information such as an ID number. We can construct a tensor z_j of dimensionality D that spans the full environment E. Positions in the tensor corresponding to an entity's position will take the feature value f, and all other positions will get a default value. Mathematically:

$$\forall x_i \in X, \forall f_j^{x_i} \neq c_{default}, \forall (k_1, \cdots, k_d) \in E$$
$$z_j(k_1, \cdots, k_D) = \begin{cases} f_j^{x_i} & k_1 = p_1^{x_i}, \cdots, k_D = p_D^{x_i} \\ c_{default} & \text{otherwise} \end{cases} \tag{3.1}$$

Because the tensor spans the whole environment, entities can be added and removed without affecting the dimensionality, only the values in the tensor.

Most environments will have multiple features for each entity. We can construct one z_j tensor for each feature. If the features represent the same semantic information, we can put them in the same tensor. We can then stack each of these z_j tensors along a new dimension to give us a tensor representation of the full state: $S_t = z_q \times \cdots \times z_{|F^x|}$. This includes all of the information in the original state, but with a fixed dimensionality $D + 1$, where D has size equal to the span of the environment and the final dimension has size equal to the number of feature types.

For 2D environments, this is like a multi-channel 2D image. Many mobile robot tasks can be considered a 2D environment. For example, RoboCup SSL is often treated as fully 2D, as the only 3D component is when the ball is kicked in the air. The navigation and positioning of the robots themselves, can be considered entirely on a 2D plane. Figure 3.1 from (a) to (b) shows an example of mapping a 5×5 grid world with 2 feature types to this representation.

Actions can be represented as a one-hot, $D + 1$ dimensional tensor. The first D dimensions have size equal to the environment, and the final dimension is equal to the number of action types: $|A_P| + |A_d|$. To select a discrete action, the agent can mark its own position in the actor tensor. To select a parameterized action, it can mark the parameter's position in the action tensor. This action space will be larger than the original action space. Actions that do not have a matching action in the original action space can be mapped to a "do nothing" action or a similar default action. Figure 3.1 part (d) shows an example of an action tensor where action a_p is selected with a parameter corresponding to object 0's position.

Here were presented a simple mapping where each entity position maps to a single location in the state tensor, and each action maps from a simple position in the action tensor. However, it is possible to map the feature information in the state to multiple positions in the state tensor, or to allow multiple action tensor positions to map to the same action in the original action space. In a 2D environment, this state can be thought of as generating an image of various shapes filled with feature value. Similarly, in the action space, it allows the agent to mark a region to indicate an action.

3.4 Policy Learning with Tensor Representations

Agents can use any RL algorithms that learns action-value functions. In this work we use the standard Double DQN algorithm [41]. However, other algorithms such as Soft Actor-Critic (SAC) with a Gumbel-Softmax distribution could also be used [39, 40, 51]. Any RL policy that can handle multi-dimensional state tensors and discrete action spaces should work. We can learn a policy network which has an output equal in size and dimensions to A_t. The value of each position in this output tensor would correspond to the action-value function value for the input state and marking that particular spot in the one-hot action tensor. Therefore, to get the action tensor that our mapping function ψ expects from the output of the policy network, we can just take an argmax. Figure 3.1 (c) shows an example Q-value tensor output. Part (d) shows how the argmax transforms this dense tensor into the 1-hot action tensor required by the ψ mapping.

3.4.1 Single Agent Tensor State-Action Training Algorithm

Algorithm 1 shows the pseudocode for training a single-agent tensor state-action space policy. We refer to this as the Tensor State-Action - Single (TSA-Single) algorithm. This algorithm follows a typical RL training loop with the additional steps of converting the states and actions between the original space and the tensor space as shown in figure 3.1.

Starting the algorithm, line 2 initializes an empty replay buffer B. Line 3 initializes parameters for the FCQN network using a standard deep learning initialization function such as Glorot initialization. [37] Line 4 begins the main training loop. Note that in practice one would usually

collect some number of random samples and add these to the replay memory before beginning the actual training to prevent over-fitting to the first few samples. Line 5 initializes the environment to a new episode and gets the starting state s in the original state space. Then for the remainder of the episode we repeat liens 6 through 17. Each iteration of the inner episode loop, we convert the current state s to our tensor state s_t through our position mapping function ϕ as shown on line 7. We then evaluate this state s_t with our current FCQN parameters θ to get a tensor action a_t as shown on line 8. On line 9 we add some exploration to this tensor action, typically with greedy-ϵ exploration. On line 10 we convert the selected tensor action a'_t back to the original action space a through our position mapping function ψ. Lines 11 through 13 are what is done in a typical RL algorithm. We execute the action, add a new transition to our replay and then sample a mini-batch of transitions. The samples we store use the original state space s because they typically require less memory. So line 14 converts every state s and next state s' in the mini-batch transitions to s_t and s'_t using the mapping function ϕ to create a tensor batch b_t. Line 15 does a standard RL policy update using whichever training algorithm one chooses. We use Double DQN in the experiments. [41] Line 16 updates the current state for the next loop and the whole process repeats.

Algorithm 1 Tensor State-Action - Single (TSA-Single) training algorithm.

1: **function** TSA-SINGLE
2: $B \leftarrow \emptyset$ ▷ Empty replay memory
3: $\theta \leftarrow$ Initialize params for π_t
4: **while** Training **do**
5: $s \leftarrow$ reset environment state
6: **for** episode length **do**
7: $s_t \leftarrow \phi(s)$ ▷ Transform to tensor state
8: $a_t \leftarrow arg\max \pi_t(\theta, s_t)$ ▷ Select action
9: $a'_t \leftarrow exploration - policy(a_t)$ ▷ Add exploration noise
10: $a \leftarrow \psi(a'_t)$ ▷ Transform to original action space
11: $s', r \leftarrow$ execute a
12: $B \leftarrow B \cup \{(s, a_t, r, s)\}$ ▷ Add transition to replay buffer
13: $b \leftarrow sample - batch(B)$
14: $b_t \leftarrow apply - \phi - to - state(b)$
15: $\theta \leftarrow update - policy(\theta, b_t)$ ▷ We use DDQN in experiments
16: $s \leftarrow s'$
17: **end for**
18: **end while**
19: **end function**

3.4.2 Masking Channels

The tensor based action space has the downside of being much larger compared to the original action space. The benefit of this additional dimensionality is that we can transfer without learning new parameters, however, exploring a large dimensional action space may still be difficult. This

is a problem that has been encountered in other domains such as Starcraft II [113]. An approach that has been used is to provide an additional input with a binary list of whether a particular action is relevant to the current state [113]. We mimic that approach in our work by providing an additional "Masking channel". Each state tensor will have an additional channel where positions corresponding to valid markable positions in the action space are marked, and all other positions take on some default value. Note this does not give away the final policy to the agent. The agent must still 1) learn to take advantage of this additional information and 2) choose the optimal action from among the marked action positions.

3.4.3 Fully Convolutional Q-Network (FCQN)

The policy network π_t could use a standard architecture of convolutional layers for the image-like state input, followed by a flattening operation and some number of FC layers. However, this representation is non-ideal for reasons including:

- FC layers can only accept fixed size inputs, so our policy will only work on a fixed size environment.
- FC layers require a flattened input and flattening will lose all of the spatial information built into our tensor representation. This makes it difficult to correlated positions in the action tensor with the state tensor, and does not allow us to bias the policy to take advantage of locality and translational invariance in an environment.
- The size of the action space is large, which will require a large FC layer with many parameters. Having lots of parameters will generally slow down training and increase the required training data.

Instead, we can adapt Fully Convolutional Networks (FCN) [61] from computer vision to represent the Q-values and our policy. We refer to this architecture as Fully Convolutional Q-Networks (FCQN). Long et al. [61] introduced a method for creating a network entirely out of convolutions and deconvolutions in order to perform dense image segmentation. Dense image segmentation requires mapping each position in an input tensor to a class. This produces an output tensor of the same size where each position's value correspond to the class value of the same position in the input image. Our tensor policy is performing a similar operation, but instead of mapping positions to discrete class values, we are mapping the positions to continuous Q-values, which are then converted into an action choice. The advantage of using all convolutions and deconvolutions for the policy network are as follows:

1. (de)convolutions do not require a flattened input
2. (de)convolutions are designed to work well for local interactions
3. (de)convolutions are designed to take advantage of translational invariance of an input
4. (de)convolutions can accept any size input and output a proportionally sized output without any change in the number of parameters or the values of those parameters

Because the tensor is never flattened, the spatial information encoded in our representation is maintained. This should make it easier for the policy to correlate positions in the action tensor to positions in the state tensor. Locality and translational invariance can allow an agent to more

easily generalize experiences across positions in the state space. Finally, because any size input can be accepted, we can also apply a single policy across changes to environment sizes. A larger environment, will be represented by a larger tensor. The same (de)convolutions can be applied to get out an action tensor proportionally sized, with no changes to the parameters of the layers themselves.

3.5 Multi-agent Policies

As discussed, our representation is designed to transfer when the number of agents in the environment varies. Therefore, if we can learn a good multi-agent policy, we can do zero-shot transfer as the number of agents varies. For domains such as robot soccer, this is a very useful property of the representation. Throughout a game of robot soccer, robots may be added or removed from the field for many reasons including rule violations and mechanical breakdowns. SSL RoboCup currently uses team sizes of 8, but plans to move to a full sized soccer team of 11 robots. If we learned a policy that was not capable of transferring across team sizes, we would have to learn policies for each combination of team sizes which would be 121 policies. Additionally, it may be more difficult to learn a policy with many robots due to the increase in states that can be encountered in a typical run of the game. By allowing for transfer, it may be possible to learn in a simpler case of a handful of robots on each team, and then scale up this policy to the full team size through transfer.

We use a simple multi-agent setup, where each agent has executes its own policy. Each agent receives its state features and constructs its tensor representation. It then evaluates its own tensor policy to pick its own individual actions. Rewards are shared across the whole team. Since the goal of this work is to focus on representation, we do not make use of any of the complex MARL approaches such as learning to communicate. We apply Double DQN to learn each agent policy just as in the single agent case. To help stabilize the policy learning and avoid issues inherent in MARL we make use of an averaging layer as described in Sukhbaatar et al. [97]. An internal hidden layer of each agent policy is averaged together to produce an additional input into each agents next hidden layer. Figure 3.2 shows an illustration of this. This gives an insight into what the other agents on the team are doing. And an averaging operation can support any number of inputs while producing a fixed size output. Therefore, transfer with no additional parameters is still possible.

The final issue is what weights to add when a new agent is added to the environment (or conversely what set of weights to remove). Instead, of requiring new sets of weights to be added or removed, we force all agent policies to share the same set of weights. Therefore, adding a new agent just means we evaluate an additional copy of the network weights on that new agents state. Similarly, if we remove an agent we just evaluate one less copy. Note, that despite the policies all sharing the same weights, we are not learning a centralized policy. Each agent still evaluates the policy on its own states and produces its own personal actions.

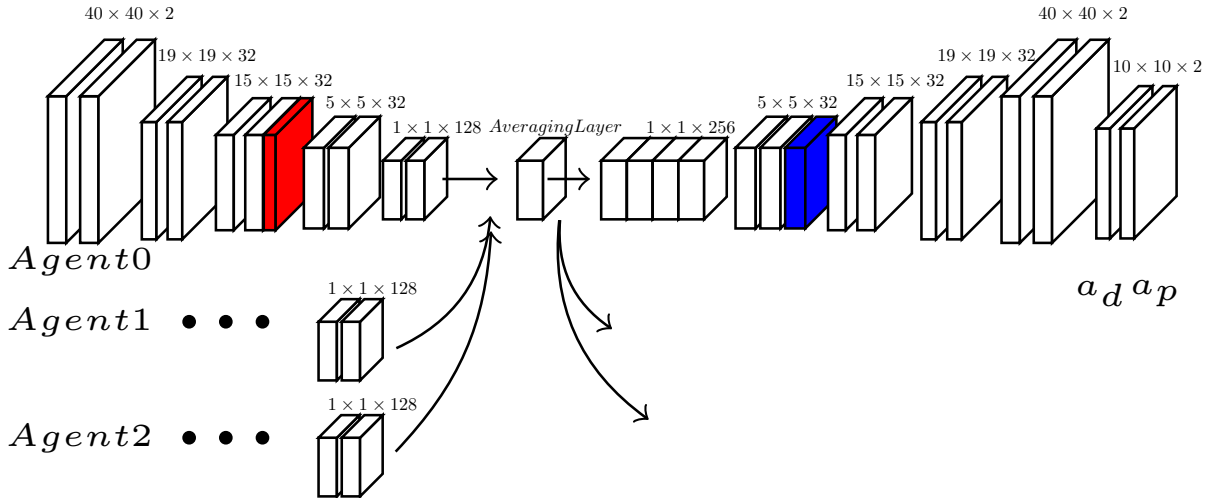


Figure 3.2: FCQN network architecture with multi-agent averaging layer.

3.5.1 Multi-Agent Tensor State-Action Training Algorithm

Algorithm 2 shows the pseudocode for training a multi-agent tensor state-action policy. We refer to this as the Tensor State-Action - Multi (TSA-Multi) algorithm. It is very similar to the TSA-Single algorithm shown in algorithm 1 but adapted to work with multi-agent procedure described above. The main differences are that for each step we have multiple states and actions (one for each agent in the environment). Additionally, we need to split the policy network into the part that computes the hidden layer inputs for the averaging layer f and the part that uses the hidden layer and average hidden layer to compute the action g. In this case $\pi_t = g(f(s_t), \bar{h})$ where $\bar{h}$ is the average hidden layer and s_t is the agent's tensor state.

Line 2 initializes an empty replay memory. In practice one will usually fill this with some random samples before starting the training loop. Line 3 and 4 initializes both halves of the network, the pre-averaging layer weights f and the post averaging layer weights g inside π_t. For each episode we start by resetting and getting the initial states for each agent on line 6. Lines 7 through 20 repeat until the end of the episode. On line 8 we convert all the agent states to tensor states using the mapping function ϕ. Line 9 computes the hidden layer just before the averaging layer for each agent. We call these outputs h and they come from the f part of the policy network. Line 10 is the averaging layer. All h from the agents are averaged to produce $\bar{h}$. Line 11 computes each agent action using that particular agent's hidden output h as well as the average hidden output $\bar{h}$. The actions for each agent may be modified for exploration on line 12. Line 13 converts every tensor action to the original action space. Line 14 executes all agent actions to get new agent states and a team reward r. Because all of the weights are shared among agents, we add all agent samples to the same replay buffer. Line 16 through 18 are standard RL procedure. We sample a mini-batch, then convert all the states in the batch to our tensor state space, and then update the policy weights. Finally 19 does some book-keeping to setup for the next iteration of the loop.

Algorithm 2 Tensor State-Action - Multi (TSA-Multi) training algorithm.

function TSA-MULTI
  $B \leftarrow \emptyset$ ▷ Empty replay memory
  $\eta \leftarrow$ Initialize shared f parameters ▷ Part of policy π_t
  $\theta \leftarrow$ Initialize shared g parameters ▷ Part of policy π_t
  while Training **do**
    $s^0, \cdots, s^k \leftarrow$ reset environment state ▷ k agent states
    for episode length **do**
      $s_t^0, \cdots, s_t^k \leftarrow \phi(s^0), \cdots, \phi(s^k)$ ▷ Transform each agent state
      $h^0, \cdots, h^k \leftarrow f(\eta, s_t^0), \cdots, f(\eta, s_t^k)$ ▷ Evaluate up to averaging layer
      $\bar{h} \leftarrow \frac{1}{k} \sum_{i=0}^{k} h_t^i$ ▷ Average each agent hidden layer
      $a_t^0, \cdots, a_t^k \leftarrow arg\max g(\theta, h^0, \bar{h}), \cdots, arg\max g(\theta, h^k, \bar{h})$
      $a_t^{0'}, \cdots, a_t^{k'} \leftarrow exploration - policy(a_t^0), \cdots$
      $a^0, \cdots, a^k \leftarrow \psi(a_t^{0'}), \cdots, \psi(a_t^{k'})$
      $(s^{0'}, \cdots, s^{k'}), r \leftarrow$ execute$(a_t^{0'}, \cdots, a_t^{k'})$
      $B \leftarrow B \cup \left\{(s^0, a_t^{0'}, r, s^{0'}), \cdots, (s^k, a_t^{k'}, r, s^{k'})\right\}$
      $b \leftarrow sample - batch(B)$
      $b_t \leftarrow apply - \phi - to - state(b)$
      $\theta, \eta \leftarrow update - policy(\theta, \eta, b_t)$ ▷ We use DDQN in experiments
      $s^0, \cdots, s^k \leftarrow s^{0'}, \cdots, s^{k'}$
    end for
  end while
end function

3.6 Simulated Empirical Results

In this section we present empirical results for training both single and multi-agent policies on three different simulated environments. Two of the environments are inspired by RoboCup SSL tasks. We also include some simulated results from a modified version of Atari Breakout in order to show the generality of the contributed representation and learning approach.

3.6.1 Environment Descriptions

We use three different environments for the experiments: `Passing`, `Take-the-Treasure`, and `Breakout`. `Passing` and `Take-the-Treasure` are grid-worlds representing abstracted parts of RoboCup robot soccer. `Passing` is a single agent environment. The agent starts with a ball. There are a number of environment controlled teammates and environment controlled opponents which move randomly around the environment. The goal is for the agent to pass the ball along a column or row of the grid to a teammate as quickly as possible without interception by an opponent. `Take-the-Treasure` is a grid-world environment inspired by Keepaway [95] that is multi-agent. It has the same dynamics as `Passing`, but this time there is a team of agents that start with the ball. The goal, like Keepaway, is to keep the ball, or in this case treasure, away from the environment controlled opponents. Opponents capture the treasure by occupying an adjacent

position to the treasure holder. Just like `Passing` the agent with the treasure can choose to pass to a teammate along a row or column. While this environment has a similar goal to Keepaway the dynamics are sufficiently different that we gave it a different name. Additionally, as will be shown in later sections because of the differences, the optimal strategy learned by the agents is significantly different. Finally, `Breakout` is a clone of Atari Breakout [67]. The difference being that our clone allows for the size of the playing environment to be modified. Changing the height changes the number of rows of bricks. Changing the width increases the number of bricks in each row as well as the allowable travel distance of agent controlled paddle.

`Passing` and `Take-the-Treasure` use the state tensor mappings described in the previous section. `Breakout` is already represented as an image, which is compatible with our tensor representation, so no modification is made to the state. Just the raw pixel representation is used. In `Breakout`, the action tensor has 3 regions that can be marked. A region centered on the paddle indicates a "do nothing" action which keeps the paddle in place. A region to the left of the "do nothing" region indicates a move left, and similarly, a region to the right indicates a move right. Figure 3.3 shows an image from the `Breakout` domain with the three action regions highlighted in different colors. The red region corresponds to the "move left" action, the green region to "stay still" and the blue region to the "move right" action.

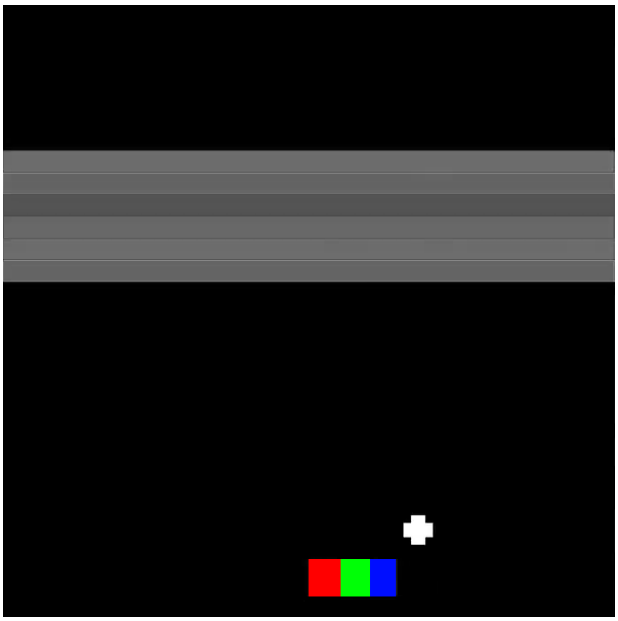


Figure 3.3: Action pixel regions for `Breakout`.

3.6.2 Hyper-Parameters and Network Architecture

We tested three different combinations of network architectures and state-action representations. As a baseline we use a standard Fully Connected layer network architecture with the original feature state space and discrete action space. To test the tensor state-action space separately from the FCQN architecture we train a Fully Connected layer network with the tensor state-action

space. Finally, we test the FCQN network architecture with the tensor state-action space. This same network is then used in all the transfer experiments and the real-robot experiments shown later.

The `Passing` FCQN network is the same as shown in figure 3.2 with the same layer sizes. Skip connects were used between all matching sized pairs of conv/deconv layers. All hidden layers use ReLU activations.

The `Passing` using state features and fully connected layers and the `Passing` network with tensor state input and FC layers both use approximately the same number of parameters as the `Passing` FCQN network. This is to remove issues where one representation outperforms another solely because it has more parameters that can be tuned. Details about the specific sizes and connections can be found in Appendix A.3.

Figure 3.4 shows the network structure used in the breakout experiments. All hidden layers use ReLU activations. Skip connections are used between matching sized convolution and deconvolutions.

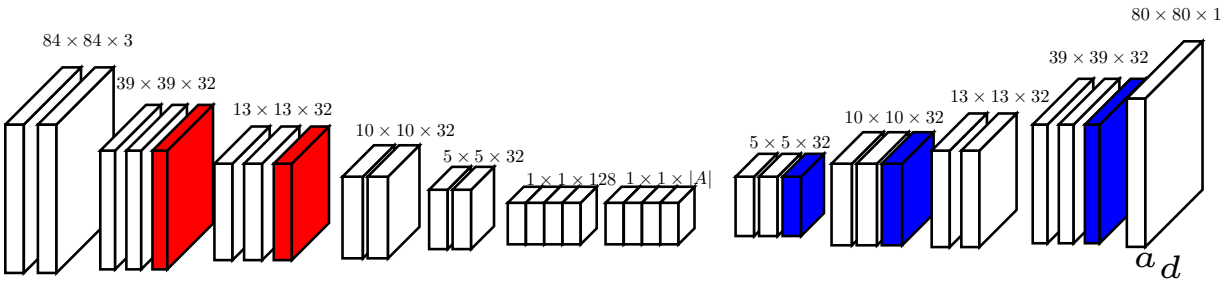


Figure 3.4: `Breakout` network structure. Red: max-pooling layers. Blue: bilinear upsampling layers.

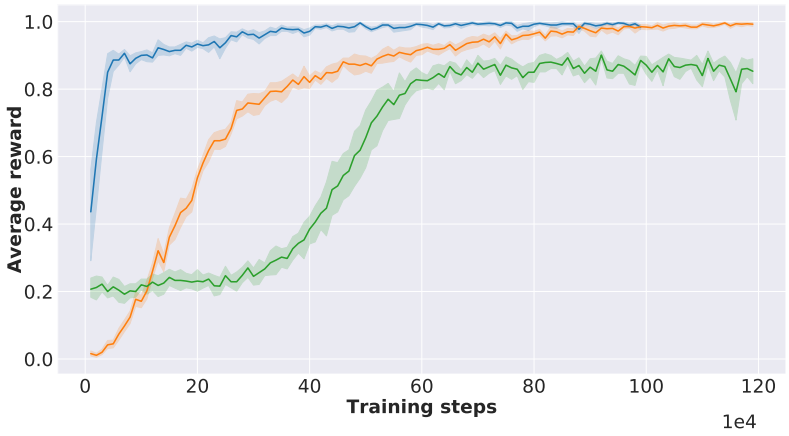
All networks were trained using the Double DQN algorithm [41] and the Adam optimizer. ϵ-greedy exploration was used during training. Appendix A.3 provides detailed numbers for the specific parameters.

3.6.3 Learning Performance

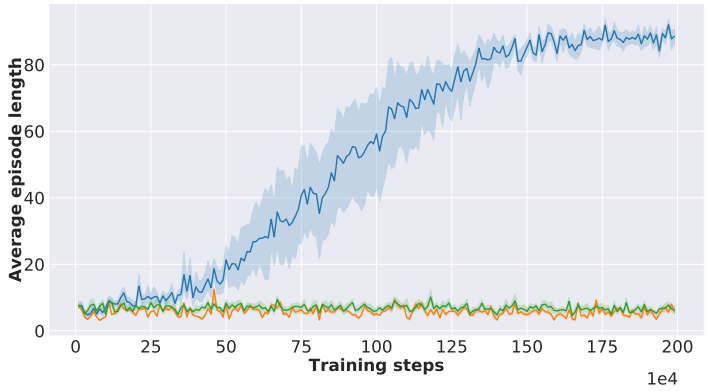
Figures 3.5a, 3.5b, and 3.5c show the training curves for our method compared to baselines. We compare our approach to a network with fully connected layers applied to the tensor representation, and a dueling network architecture [114] using a state vector and discrete action space. For each evaluation point, ϵ was set to zero and 100 episodes were run, and the average return was calculated. This was repeated from scratch 10 times. The solid line shows the average across the 10 runs, and the shaded region shows the minimum and maximum performance across the 10 runs. Both the FCQN network and the fully-connected network have approximately 3.5 million parameters, so differences in performance cannot be accounted for by the amount of free parameters.

In both `Passing` and `Take-the-Treasure` the FCQN with the tensor representation converges significantly faster than either the tensor representation with FC layers or the standard

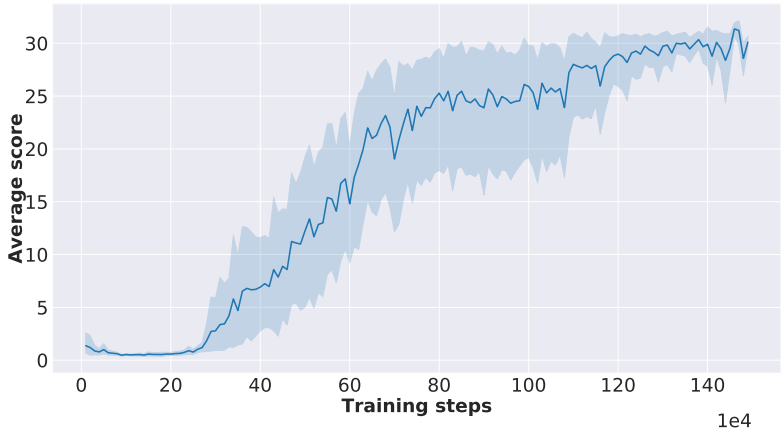
representation with a dueling architecture. In fact, the other representations fail to learn a meaningful policy on `Take-the-Treasure` within the allocated training steps. This demonstrates that the combination of FCQN and tensor representation can lead to speed-ups in environments where spatial information, locality, and translational invariance are important to the task. Note that for `Passing`, a reward of 1.0 is the best possible outcome, showing our representation does allow for optimal policies.



(a) `Passing` training curve



(b) `Take-the-Treasure` training curve



(c) `Breakout` training curve

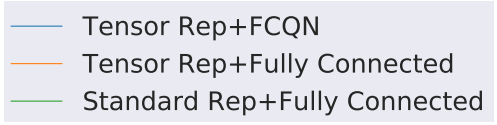


Figure 3.5: Average training performance.

`Take-the-Treasure` was trained with 3 agents vs 3 agents on a 10×10 environment. The opponents learn a coordinating policy, where the treasure holder runs from opponents, while teammates form walls or surround the holder to block opponents. This wall building and surrounding behavior is likely optimal. As if the teammates cannot block off all routes to the treasure-holder they can hold the treasure indefinitely. Thus passing is actually less of an optimal strategy in most cases for this environment.

Figure 3.6 shows parts of a sequence of the trained `Take-the-Treasure` policy being executed. Green cells are empty field. Blue cells are teammates. Yellow cells are opponents. The red cell is the treasure holder. Figure 3.6a show the initial configuration from this run. The treasure holder, teammates and opponents are initialized at random positions. Figure 3.6b shows the state after a few steps in the environment. We can see the yellow opponents closing in on the red treasure holder. We can also see the two blue teammates forming a small wall that blocks the direct path of the yellow opponents below. Figure 3.6c shows another state a few steps later where the treasure-holder is being chased around the wall formed by the teammates. This being chased around the wall behavior is continued in figure 3.6d. Sometimes, the yellow opponents manage to out-maneuver the teammates, which case the wall must shift and change shape. We can see a different wall shape in figure 3.6e from later in the same episode.

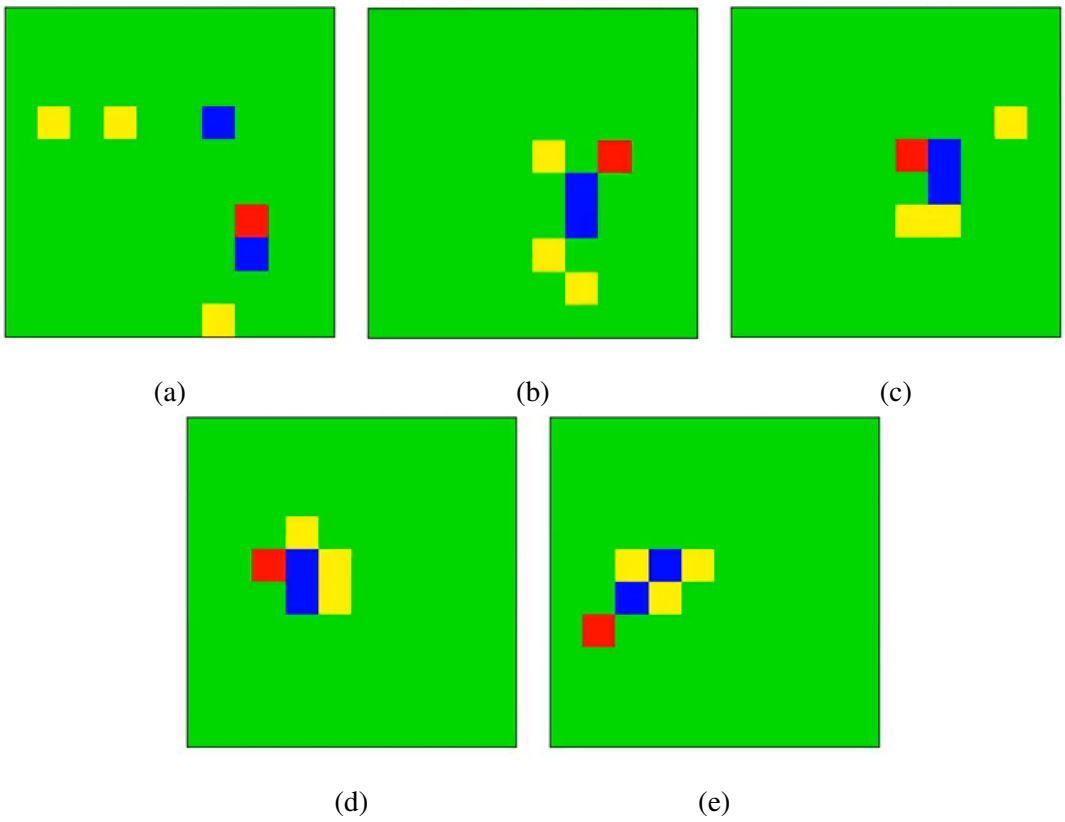


Figure 3.6: Images from execution of `Take-the-Treasure` policy operating with training environment size and team sizes.

`Breakout` takes the longest to begin showing improvement. This is likely because the

dynamics in the environment are more complicated than in the other environments. However, by approximately 1 million steps, the qualitative performance becomes good. The agent will clear approximately 25 blocks before losing a life. More training could potentially further improve this policy.

Figure 3.7 shows a sequence of frames from an episode of the trained `Breakout` policy. Like the results from the original Atari DQN paper we see a similar policy of "tunneling". [67]. Figure 3.7a shows the initial start of the episode. In figures 3.7b and 3.7c shows the agent building the tunnel by breaking blocks along the right column of the screen. Figure 3.7d and 3.7e shows the agent taking advantage of the tunnel by bouncing the ball between the bricks and the ceiling to quickly clear a large section with minimal risk. Finally 3.7f shows the state of the world after the advantage of the tunnel is gone and the agent needs to carefully clear the remaining blocks.

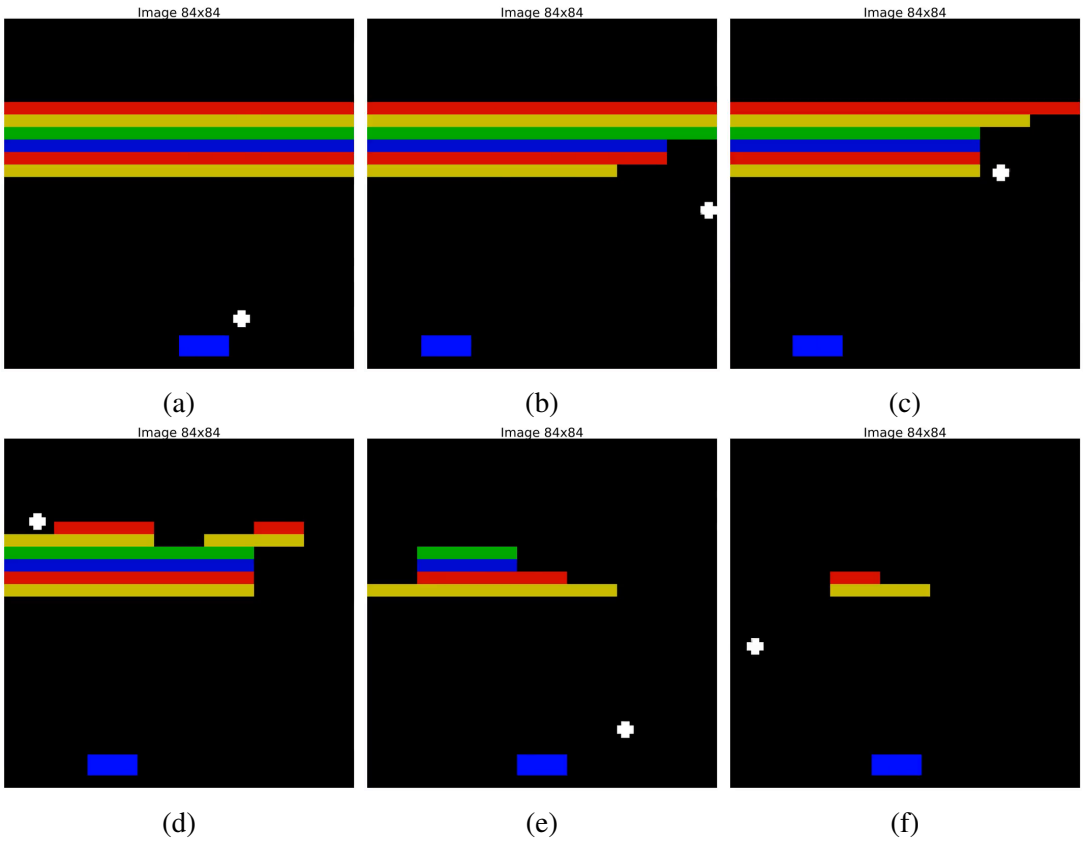


Figure 3.7: Example sequence from `Breakout` policy operating on same size environment as training environment.

3.6.4 Zero-shot Transfer Across Team Sizes

We trained one `Passing` network with a team size of 3 vs 3 opponents on a 20x20 environment. Figure 3.8 shows the average reward achieved by the policy as it transfers across team and opponent team sizes, as well as a version normalized to the original team-size performance. Figure 3.8a shows the absolute performance for each team size combination, where brighter is

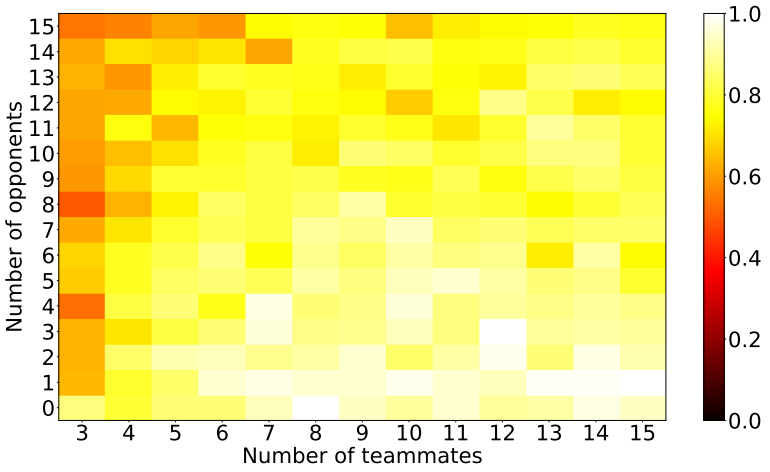
better. The absolute performance remains relatively consistent and close to the max possible 1.0 across the team size combinations tested. Only when there are large number of opponents, such that few passing opportunities occur, do we see a major difference in performance.

Figure 3.8b shows the same data, but normalized with respect to the reward achieved on the original team size of 3 vs 3. This makes it easier to see small changes in the performance across transfers. Here team size combinations with value equal to 1.0 have the same performance as the original team size combination. This is represented by a white color. As performance increases relative to the original team size combination performance, the color becomes more green and the values increase from 1.0. As performance decreases relative to the original team size combination performance, the color becomes more purple and the values decrease from 1.0. We can see that overall the performance is better than on the original team size. This is expected as more teammates means more passing opportunities. The exceptions happen on the part of the grid where the number of opponents is much higher than the number of teammates, and even there the performance is mostly equal to the original performance, with only a few combinations having a drop in performance of about 20%.

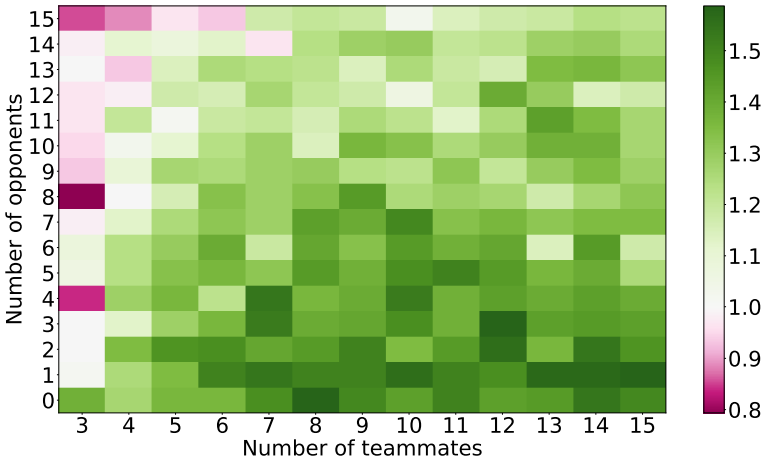
We trained one `Take-the-Treasure` network with 3 agents and 3 opponents on a 10x10 grid environment. Figure 3.9 shows the average reward achieved by the policy as it transfers across team sizes and opponent team sizes, as well as the same data but normalized to the original team size combination's performance. Figure 3.9a shows the absolute performance for each team size combination, where brighter is better. We see good transfer performance, when the team sizes are equal or the agent team is larger. This is expected, as 1) the randomly chosen starting state is less likely to have the treasure holder already surrounded by opponents and 2) it is easier for more opponents to out maneuver the teammates.

Figure 3.9 shows the same data but normalized to the performance on the original team size of 3 vs 3. Cells with performance equal to the original team size have a white color and a value of 1.0. Team size combinations that perform better than the original team size combination have a value greater than 1.0 and are more green the better they are. Team size combinations that perform worse than the original team size combination have a value less than 1.0 and are more purple the worse they are. We see that when teammates greatly outnumber the opponent team size the performance is better, as expected. Similarly, when close to equal, the performance remains the same as in the 3 vs 3 case. And when opponents outnumber teammates the performance drops, with more severe drops the greater the team size imbalance.

Figure 3.10 shows a sequence of images taken from the execution of the `Take-the-Treasure` policy trained with 3v3 to an environment that is 4v1. Green represents empty field, blue teammates, yellow opponents, and red the treasure holder. Figure 3.10a shows the initial state of this episode. Figure 3.10b shows the closest teammates (in blue) starting to surround the treasure-holder (in red) to protect it from the yellow opponent. Figures 3.10c, 3.10d, 3.10e show the teammates forming blocking walls while the treasure-holder gets chased around these walls by the yellow opponent. Finally figure 3.10f shows an example where the teammates surround the treasure-holder, minimizing the available directions the treasure-holder can be tagged from.

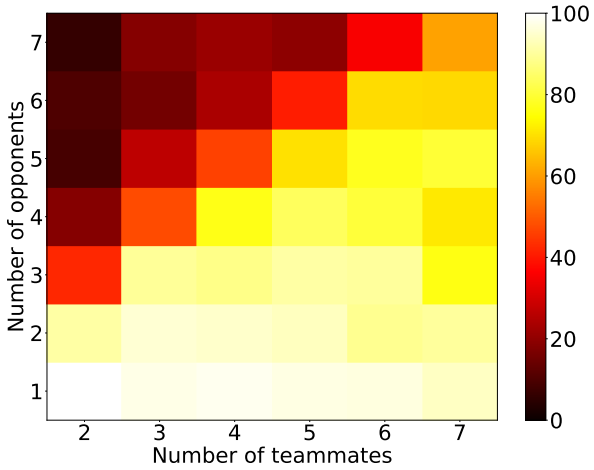


(a) `Passing` transfer as team sizes vary

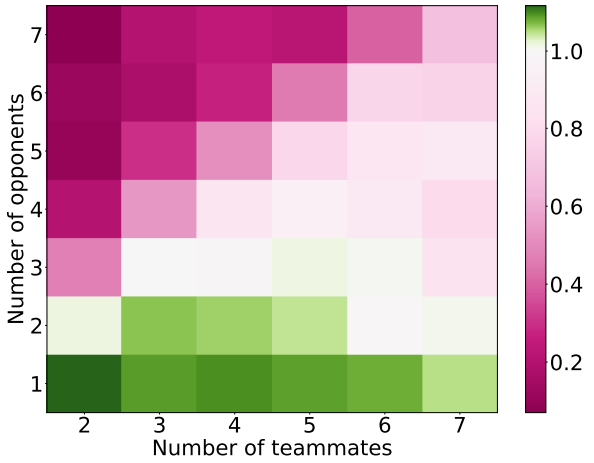


(b) `Passing` transfer as team sizes vary, normalized to performance on original team size

Figure 3.8: Absolute and normalized performance of transfer across team sizes for `Passing`

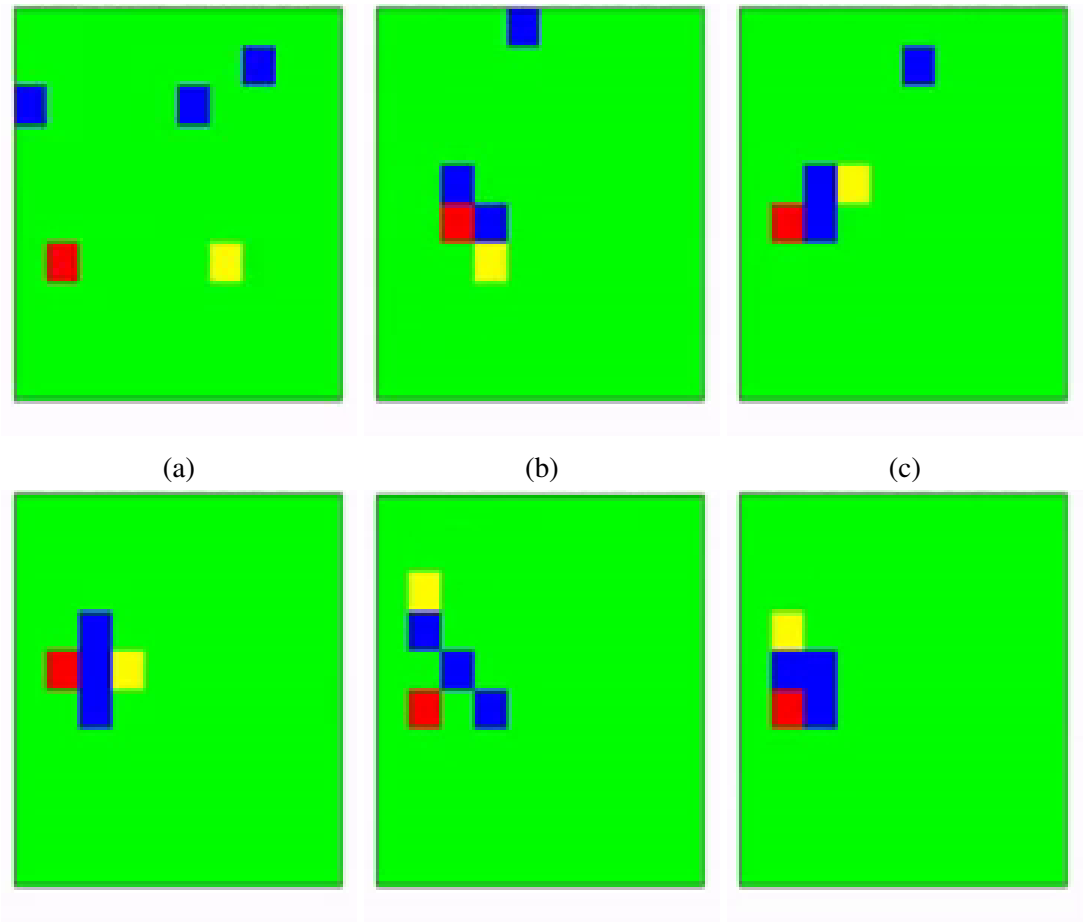


(a) `Take-the-Treasure` transfer as team sizes vary



(b) `Take-the-Treasure` transfer as team sizes vary, normalized to performance on original team size

Figure 3.9: Absolute and relative performance of transfer across team sizes for `Take-the-Treasure`



(a) (b) (c)

(d) (e) (f)

Figure 3.10: Example sequence executing `Take-the-Treasure` policy transferring to a different team size.

3.6.5 Performance of Transfer vs Trained from Scratch

While the transferred policy has similar performance before and after transfer, it may actually perform poorly vs a policy trained directly with the new team size. We trained policies from scratch on different team sizes for 500,000 steps and compared this policy performance vs the zero-shot transfer performance of the policy from the previous section. We evaluate the relative performance defined as $(R_T - R_S)/R_S$ where R_T is the average reward/episode length for the transferred policy and R_S is the average reward/episode length for the trained from scratch policy.

Table 3.1 shows the statistics for relative transfer performance on `Passing` and `Take-the-Treasure`. `Passing` was evaluated for every team size combination from Figure 3.8a. `Take-the-Treasure` was evaluated across 7 different team size combinations.

	Relative Performance		
	Min	Max	Mean $\pm$ Std Dev
`Passing`	-0.25	0.2	-0.03 $\pm$ 0.07
`Take-the-Treasure`	-0.11	0.26	0.14 $\pm$ 0.31

Table 3.1: Relative performance of policies transferred to different team sizes vs policies trained from scratch with those team sizes.

`Passing` transferred performance is only slightly worse on average, with a small standard deviation. `Take-the-Treasure` has a larger standard deviation, but on average the transferred policies actually slightly outperform the from scratch policies. This is likely, because training in a smaller environment and then transferring to a larger environment allows the agent to explore and learn in a smaller state and action space. Thus requiring less samples for training, yet retaining the performance when moving to a larger environment.

3.6.6 Zero–shot Transfer Across Environment Sizes

Table 3.2 shows the average transferred policy performance as environment size changes. Each new size evaluated the transferred policy over 100 independent episodes. `Passing` was trained on a 20x20 grid and then evaluated as height and width were varied independently from 20 to 40 with steps of 5. `Take-the-Treasure` was trained on a 10x10 grid and transferred to grids of different heights and width varied independently from 10 to 20. `Breakout` was trained on the standard 84x84 pixel image and transferred to environments of different heights and widths varying from 84 to 120 in increments of 6 pixels.

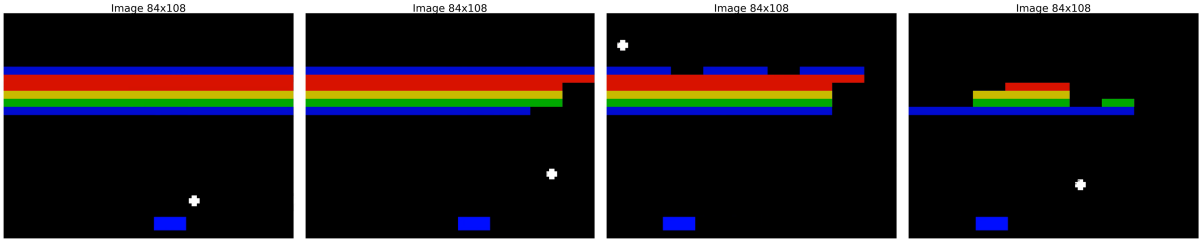
Just like when transferring across number of agents in the environment, we see that transferred performance is relatively consistent as environment size changes. This is despite never seeing any training data from the new environment sizes.

Figure 3.11 shows a sequence from an episode where the `Breakout` policy trained on an 84x84 image is applied zero-shot to an 84x108 image. This can be compared to the sequence in figure 3.7 to see that despite changes in the environment size the behavior is similar. Figure 3.11a shows the initial state. Just like in the original sized environment, the agent employees a tunneling strategy. Figure 3.11b shows the agent building this tunnel. Figure 3.11c shows the agent

	Mean $\pm$ Std Dev	Unit
`Passing`	0.89 ± 0.03	Reward
`Take-the-Treasure`	84.67 ± 6.55	Episode Length
`Breakout`	19.34 ± 7.98	Blocks per life

Table 3.2: Performance of transfer across environment sizes.

taking advantage of the tunnel to break a lot of bricks. Figure 3.11d shows the aftermath of using this tunnel.



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

Figure 3.11: Sequence of images taken from a `Breakout` policy trained on 84x84 images running on an image of 84x108.

3.7 Applications to Real Hardware

In the previous section, we showed that using the contributed tensor state-action space and FCQN network, it is possible to learn well-performing policies in a variety of simulated domains. In this section, we will show that despite the abstract nature of the simulated domains and the representation, it is possible to apply learned policies to real robot hardware. We utilize the simulation trained `Take-the-Treasure` policies with the RoboCup SSL robots and camera system described in chapter 2.

3.7.1 Real-world State Action Mappings

We use the policies trained in simulation with no additional learning. We run on a field size of approximately 5m by 3m, although the exact field dimensions can be changed thanks to the transferrability across environment sizes of the representation. Robot positions detected by the camera system are discretized into a grid with cells being approximately the diameter of a robot (i.e. 18cm by 18cm), learning to a 28×17 tensor state. Orientation is ignored as the robots can move omni-directionally, and orientation had no affect in the simulated environment. The environment tracks the feature of which robot controls the treasure and which team each robot belongs to. This information is also added to the tensor state in the same way described in the previous sections.

The move actions are translated to continuous desired positions in real-world coordinates at the center of the discretized field grid cell. These coordinates are used as set points of a PID controller running at 60Hz using the raw vision system detections. Passing the treasure updates the internal environment state of which robot currently has the treasure.

New decisions are made by the agents and opponents at a fixed 5Hz regardless of whether the PID controller has reached the desired set-point chosen at the last decision time. In simulation actions are guaranteed to complete and always succeed (assuming the action is allowed by the environment transition function). However, the real robots may not finish executing and due to noise in the actuation and sensing, may not achieve the desired positions. Because the policy is deterministic, new actions will only be selected if at least one robot has changed positions or a successful treasure pass action has been executed.

3.7.2 Empirical Results

We ran 30 runs of 3 vs 3 robots collecting the number of steps taken in each episode. Like in simulation each episode starts with random initial positions of all the robots. Table 3.3 shows the results of these experiments compared to simulation results.

	Mean $\pm$ Std Dev
Real Robot	71.2 ± 17.7
Simulation	76.13 ± 15.6

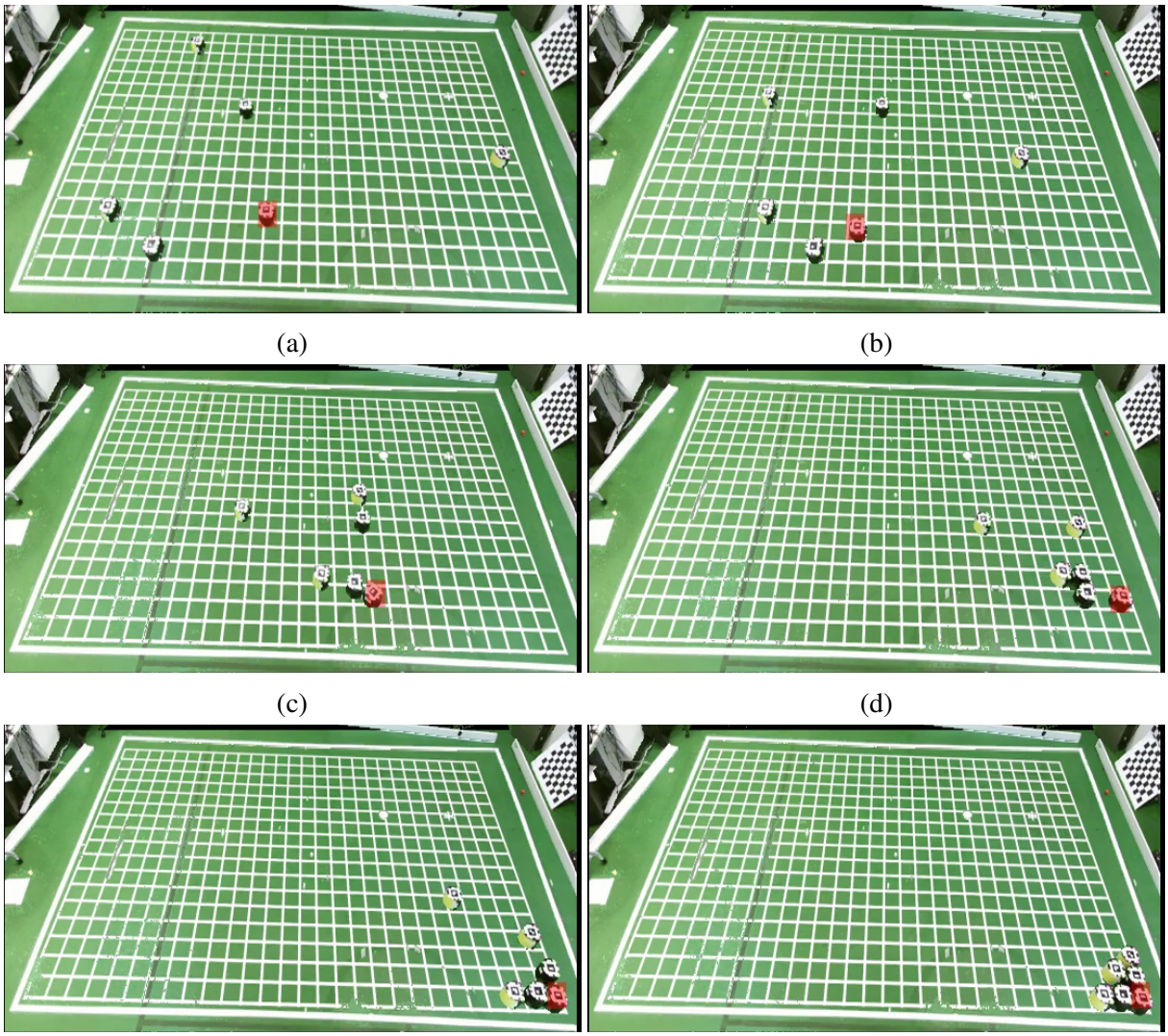
Table 3.3: Real-robot `Take-the-Treasure` policy performance. 3 vs 3 robots on a 28×17 grid corresponding to roughly a 5m by 3m real world environment.

We can see that despite the policies only training in simulation with perfect action execution and no noise on the input state, the policies have only a slight drop in performance when executed on the real robot hardware. This is despite noise in the computer vision detections, imperfect action execution and asynchronous execution of the agent actions.

Figure 3.12 shows a sequence of frames from the simulation trained `Take-the-Treasure` policy executing on the real robots. Note, that this is also an example of transfer across environment size as the policy was trained on a 10x10 grid, but the execution here is on a 28x17 grid. The grid lines showing approximately locations of the cells was added after the video was taken. The robot highlighted red currently has the treasure. The robots with yellow backs are opponents and the plain black robots are teammates.

Figure 3.12a shows the initial configuration for this episode. The grid is edited in for visualization purposes. The red square marks the current treasure-holder. The robots with yellow color on the back represent opponent robots and the black robots are teammate robots. Figure 3.12b shows the teammates moving in to protect the treasure-holder while the opponent robots close in from above and to the right. Figures 3.12c and 3.12d show the team and the treasure-holder moving towards the bottom right corner. Note the small 2 robot wall formed by the teammate robots that is blocking one of the opponents. Eventually the robots reach a configuration where the treasure-holder is perfectly protected. Figure 3.12e shows this configuration, with the treasure-holder in the bottom right corner and the teammates protecting tags from above and to the left.

For the rest of the episode the robots remain in the configuration seen in figure 3.12f despite the yellow robots pushing against the teammates to try and get to the treasure-holder.



(a) (b) (c) (d) (e) (f)

Figure 3.12: Example sequence of `Take-the-Treasure` policy trained on 3v3 team on a 10x10 grid executing with 3v3 real robots on a 28x17 grid.

3.8 Summary

This chapter presented the thesis contribution of tensor state-action spaces, a representation that is designed up front to facilitate transfer across different number of agents and objects in an environment. The chapter also describe the FCQN network architecture, which combined with the tensor state-action representation allows for transfer across environment sizes. By designing such a representation for states, actions and the policy network, this thesis has provided a new

method of increasing the adaptability of learned policies. We showed empirically that such techniques work well for environments where positioning is important, and that despite the abstract nature of the representation is can work on real hardware and sensors.

Chapter 4

Auxiliary Task Learning with Multiple State Representations

As shown in chapter 2, one shortcoming of existing Deep RL approaches is the large amount of training time and samples required to converge to a reasonable policy. This is especially true when using sparse rewards and high-dimensional state and action representations. The approach introduced in chapter 4, greatly improves a policies adaptability to specific types of changes in the environment, but at the cost of making both states and actions high-dimensional. In this chapter, we introduce an extension to the Scheduled Auxiliary Controller (SAC-X) framework that allows for learning with auxiliary tasks that differ not only by reward function but by state representation [89]. We show how such an approach allows us to quickly learn good policies even with high-dimensional states such as raw camera images. We begin by introducing the SAC-X framework and the thesis contribution. We discuss how auxiliary tasks are chosen. We then demonstrate our approach on a simulated RoboCup SSL task. Finally, we show that our approach works directly on real hardware with image sensors. We more fully introduce the Ball-in-a-Cup with a robot arm domain, and show empirical results for such a task in both simulation and from learning directly on real hardware.

4.1 Problem Description

One common strategy for minimizing the amount of training data required for RL is to carefully curate the information used to represent the state of the MDP into so-called features. These features — e.g. positions, orientations, and velocities of the robot links and important objects — can simplify the learning problem since: 1) learned policies can use less parameters to represent the policy and 2) learned policies do not need to learn to recognize things if interest in a high-dimensional state space. Alternatively, one can try to learn with a high-dimensional state and action space such as the tensor state-action space from chapter 3. In this case the number of states and actions to explore is greatly increased which can increase the amount of training data required. Additionally, one may also learn directly with high-dimensional sensors such as cameras. In this case the agent must learn to ignore spurious signals and distractors in order to converge to a robust, well-performing policy. In both of these high-dimensional cases, the

amount of required training data can be much larger than when training from a distilled set of task-specific features.

While feature based representations can have a sample complexity advantage, as we showed in chapter 3, this can come at the cost of an inflexible policy. Using the contributed tensor state-action space, the learned policies can adapt to changes to the number of agents and objects and changes to the size of the environment. Even more important than flexibility, when learning with real robots, computing or extracting a distilled set of features can be computationally expensive, difficult and can require additional sensor arrays and pre-processing pipelines. Even when it is possible to extract these features during training, it is often desirable to learn a policy that does not require these additional constraints. This allows the agent to be deployed in other locations and scenarios with minimal setup. For example, in the Ball-in-a-Cup with a robot arm task, described later in this chapter, we make use of an object tracking system during training. Such a system requires additional specialized cameras and markings. If such an apparatus was also required at test time, the scenarios and environments where the robot can apply this policy is severely restricted.

In this part of the thesis contribution, we introduce a method that can simultaneously learn control policies with different state spaces, from scratch in both simulation and on real-robot hardware for dynamic tasks. In this case, both a feature-based representation and a high-dimensional set of camera images are used at train time, but at test-time the policy uses only the camera image representations. The thesis method extends the multi-task RL framework of Scheduled Auxiliary Controller (SAC-X) [80] to scenarios where tasks differ not only in rewards but also in state space [1]. The approach is based on the following assumptions:

1. An extended set of observations is available to the agent at training time, including raw sensor data (e.g. images), proprioceptive features (e.g. joint angles), and "expensive" auxiliary features (e.g. object positions) given via an object tracking system
2. At test time, the expensive auxiliary features are not available, requiring the policy to rely only on raw sensors, proprioceptive features and any internal state variables.
3. Control policies from features will converge faster than policies learned from raw-sensor data, which will provide a guiding signal for learning the vision based policies
4. A subset of the observations provided to the learning agent forms a state that is sufficient for an optimal policy to be found.

Using these assumptions, our method was developed to have the following properties:

1. Like SAC-X our approach allows joint learning of a control policy for a main task and multiple auxiliary tasks
2. Our method allows for joint learning of vision and feature based policies for all tasks
3. Through a special network architecture, all of the learned policies share a set of features (i.e. they share hidden layers) in order to enable accelerated learning
4. Executing feature based skills during training improves the convergence of vision based control policies

[1] See section 5.2.1 for more background on alternative auxiliary task frameworks.

We first apply our approach to a simulated navigation task using the RoboCup SSL robots. This demonstrates that the approach works on a simple domain and that it allows for speed-ups in simulation. Following this, we demonstrate this approach on the more challenging Ball-in-a-Cup task using a robot arm. The approach is able to learn this task from scratch, on a real-robot, with no imitation data in approximately 28 hours of training on a single robot [2].

4.2 Training and Network Approach

This section describes in detail how the thesis contribution works. We first give an overview of the notation used in this section. We then explain how to expand the SAC-X framework to allow auxiliary tasks with different state spaces. We explain how policies are evaluated and updated. Finally, we discuss how our technique combines nicely with the asymmetric actor-critic technique [74].

4.2.1 Background and Notation

We assume that the agent is using a model-free reinforcement learning algorithm. For the remainder of this section we will use probabilistic policies in our notation as probabilistic policy learning has been shown to help learn more robust policies as well as improve exploration. [39, 40] However, our approach should apply to deterministic off-policy learning methods with minor notational changes. More specifically the agent is trying to find a policy $\pi_\theta^{\mathcal{M}}(a|s)$ where π is the policy for MDP $\mathcal{M}$ parameterized by parameters θ, a is an action, and s is a state. The goal is for the agent to solve the maximization problem: $J(\theta) = \mathbb{E}_{p(s)}\left[Q_{\mathcal{M}}^{\pi}(s,a)|a \sim \pi_\theta^{\mathcal{M}}(a|s)\right]$ where the action-value function (or Q-function, see section 2.2.4) is defined by $Q_{\mathcal{M}}^{\pi}(s_t, a_t) = r_{\mathcal{M}}(s,a) + \mathbb{E}_{\pi_\theta^{\mathcal{M}}}\left[Q(s_t,\cdot)|s_{t+1} \sim p(\cdot|s_t,a_t)\right]$.

In this case, we are not interested in solving a single policy π_θ for a single MDP $\mathcal{M}$. We also want to learn policies for a set of auxiliary rewards $\{r_{\mathcal{A}_1}(s,a), \cdots, r_{\mathcal{A}_K}(s,a)\}$, where like in the SAC-X paper [80] each of these auxiliary reward functions define a set of auxiliary MDPs $\mathcal{A} = \{\mathcal{A}_1, \cdots, \mathcal{A}_K\}$. Unlike SAC-X, in our approach these auxiliary MDPs may differ not only in reward function, but also in the state-space. We want to learn not only the π_θ for the main task, but also a $\pi_\theta^{\mathcal{A}_i}(a|s)$ for each of the auxiliary tasks. Like SAC-X, these auxiliary policies can be called intentions, because they are intentionally executed during training. The goal is for these auxiliary policies to entice the agent into exploring interesting regions of the state-action space. And in our approach, the auxiliary policies collect good training data for more complex representations by executing the more quickly learned policies in the feature state-space.

To collect data for the learning, an intention is randomly chosen and executed for some fixed number of steps. Such a random execution of intentions is referred to as SAC-U in the original paper [80]. During execution a reward vector $r(s,a) = [r_{\mathcal{M}}(s,a), r_{\mathcal{A}_1}(s,a), \cdots, r_{\mathcal{A}_K}(s,a)]$ is saved for each transition. All data collected is put into a replay buffer $\mathcal{B}$. An off-policy learning algorithm draws data from this buffer in order to update all of the different policies. In the experiments later in this chapter we utilize two different policy learning algorithms: Stochastic

[2]Training time ignores the episode reset time, which due to entanglement of the string adds an overhead of 20-30%.

Value Gradients (SVG-0) [44] with retrace [69] and Soft Actor-Critic (SAC). However, any off-policy, probabilistic policy learning algorithm can be used.

4.2.2 SAC-X with Different State Spaces

We extend the SAC-X framework to allow the tasks to differ not only by reward, but also by state-space. The main policy uses the state space desired at test time. In the case of RoboCup SSL, this might be the raw images from the vision system as well as any proprioceptive features. Whereas, auxiliary tasks may utilize additional features available only at training time, such as the information provided by the vision detection system that processes the raw images to extract features like robot positions.

Let S^σ be a set of all measurable signals at time t that make up an over specified full state s_t during training as shown in equation 4.1.

$$s_t = S^\sigma(t) = \left\{o_t^0, o_t^1, \cdots, o_t^n\right\} \tag{4.1}$$

Where o_t^i is an observed measurement at t. These observations may include proprioceptive features, features from external tracking systems, and "raw" sensor data such as camera images. Proprioceptive features may be readings from internal sensors such as wheel encoders from the SSL robots or joint angles from a robot arm. External features may include position and velocity estimates from a motion capture system or processing computer vision tags from the raw images. In a standard setup, a minimal subset of these observations would be used to construct the MDP state in order to make the learning process easier. For example, the state could be composed of only the proprioceptive features and the positions detected by the SSL vision system. Or we could construct a state from the raw camera images and the proprioceptive features. We consider the case where S^σ can be decomposed into multiple, non-overlapping subsets of observations:

$$\mathcal{S} = \left\{S_i^\sigma, S_j^\sigma, \cdots | S_i = \left\{o^k, \cdots\right\} \wedge S_j^\sigma = \left\{o^l, \cdots\right\} \wedge S_i^\sigma \subseteq S^\sigma \wedge S_j^\sigma \subset S^\sigma \wedge S_i^\sigma \cap S_j^\sigma = \emptyset\right\} \tag{4.2}$$

Without loss of generality, for the remainder of this explanation we consider the following three feature sets:

$S^\sigma_{\textbf{proprio}}$ Contains internal robot sensor and state information such as SSL wheel encoder values or a robot arm joint positions. Other internal data such as previous time-step actions could also be included here.

$S^\sigma_{\textbf{features}}$ Contains information extracted from raw sensor data, that is known to be needed for the task. For SSL tasks this might include estimated positions and velocities of the robots and the balls as detected by the external vision system and computer vision tags.

$S^\sigma_{\textbf{image}}$ Contains the raw images collected from the cameras. These are the same images may have been processed to produce some of the features in $S^\sigma_{\text{features}}$, but there is no restriction that they be the same image sensors. Also, in other tasks other high-dimensional sensors may be more appropriate such as high-resolution depth scans.

These are the key groups necessary to explain the approach. However, different groupings may be more appropriate depending on the set of auxiliary tasks and the main task being learned.

Using these features sets we associated a fixed masking or filter-vector with each of the constructed sets S_i^σ, given by an index vector:

$$S_i^{filter} = \left[\mathbb{I}^i(task_0), \mathbb{I}^i(task_1), \cdots, \mathbb{I}^i(task_N)\right] \tag{4.3}$$

The indicator function $\mathbb{I}^i(task_k)$ is defined on a per-task basis and returns 1 if this state set should be enabled for task k and 0 otherwise. We define two filter vectors one for the action-value critic — $S_{Q,i}^{filter}$ — and one for optimizing the policy — $S_{\pi,i}^{filter}$. This allows different combinations of feature sets for different tasks between the actor and the critic

4.2.3 Policy Evaluation

The learned action-value critic function $\hat{Q}_\mathcal{T}^\pi(s_t, a_t; \phi)$ for $T \in \{\mathcal{M}, \mathcal{A}_1, \cdots, \mathcal{A}_K\}$ with parameters ϕ is represented by a feed-forward neural network. The network is defined as:

$$\begin{aligned}
&\hat{Q}_\mathcal{T}^\pi(s_t, a_t; \phi) = f([\mathbf{e}_1 \odot \mathbf{g}_{1,:}, \mathbf{e}_2 \odot \mathbf{g}_{2,:}, \mathbf{e}_3 \odot \mathbf{g}_{3,:}]; \phi_\mathcal{T}), \\
&\mathbf{g} = [g_{\phi_p}(S^\sigma_{\text{proprio}}(t)), g_{\phi_f}(S^\sigma_{\text{features}}(t)), g_{\phi_i}(S^\sigma_{\text{image}}(t))]^T, \\
&\mathbf{e} = [\mathbb{I}^{\text{Q,proprio}}(\mathcal{T}), \mathbb{I}^{\text{Q,features}}(\mathcal{T}), \mathbb{I}^{\text{Q,image}}(\mathcal{T})]
\end{aligned} \tag{4.4}$$

Here $\mathbf{g}$ is a concatenated output of three function approximators, each of which are themselves feed-forward neural networks with parameters ϕ_p, ϕ_f, and ϕ_i respectively. Each of these feed-forward networks is applied to the state-features from each feature group. $\odot$ denotes an element wise multiplication, subscripts denote element (in the case of $\mathbf{e}_i$) and row-access (in case of $g_{i,:}$) respectively. f denotes the final output layer, and the full parameters of the network are given by $\phi = \{\phi_\mathcal{M}, \phi_{\mathcal{A}_1}, \cdots, \phi_{\mathcal{A}_K}, \phi_p, \phi_f, \phi_i\}$. With this setup, only features that are within an activated feature set for task i are utilized by the network f. This allows for joint learning of the g-networks across multiple tasks, while the output function f can specialize on a per-task basis, where a task is a combination of a reward function and a state-space. By specializing per task, rather than reward, the network can adjust the policy to account for differences in the observation types. For example, part of the environment may be very noisy in one observation type and not noisy in another. Learning is performed the same as in Riedmiller et al. [80] by regressing with retrace [69] targets:

$$\begin{aligned}
&\min_\phi L(\phi) = \sum_\mathcal{T} \mathbb{E}_{\tau\sim\mathcal{B}}\left[\left(\hat{Q}_\mathcal{T}^\pi(s, a; \phi) - Q^{\text{ret}}\right)^2\right], \text{with} \\
&Q^{\text{ret}} = \sum_{j=i}^{\infty}\left(\gamma^{j-i}\prod_{k=i}^{j} c_k\right)\left[r_\mathcal{T}(s_j, a_j) + \delta_Q(s_{j+1}, s_j)\right], \\
&\delta_Q(s_i, s_j) = \gamma\mathbb{E}_{\pi^\mathcal{T}_{\theta'}(a|s)}[Q_\mathcal{T}^\pi(s_i, \cdot; \phi')] - Q_\mathcal{T}^\pi(s_j, a_j; \phi'), \\
&c_k = \min\left(1, \frac{\pi^\mathcal{T}_{\theta'}(a_k|s_k, \mathcal{T})}{b(a_k|s_k)}\right),
\end{aligned} \tag{4.5}$$

Here b are recorded policy probabilities under which trajectory $\mathcal{T}$ was recorded and and ϕ' denotes parameters of a target network which is copied from current parameters θ using standard target network update procedures.

The above describes the general approach, however, the specific policy optimizations may differ depending on the off-policy algorithm used for learning. Additionally, aspects such as the c_k value may or may not be present depending on the use of techniques such as retrace [69].

Algorithm 3 shows the pseudo-code for the actor using our multiple state representation SAC-X approach. We refer to algorithm as Multi-State SAC-X (MS-SAC-X) Actor. This algorithm will run in a separate process from the Learner shown later. Multiple of these actors can run simultaneously to parallelize exploration like in other works. [27] The actor takes four parameters: number of trajectories N, number of steps per episode T, scheduler period ξ, and number of tasks K. Each trajectory starts by fetching the latest actor network parameters from the learner shown on line 3. Line 4 initializes an empty trajectory buffer which will be sent to the learner after the episode finishes. Line 5 resets the environment and gets the initial state. Lines 6 through 17 repeat until the episode finishes. At each episode step we check if the current time-step t is a multiple of the scheduler period ξ. If so then on line 8 we sample a new task $\mathcal{T}_i$ uniformly from the K auxiliary tasks and the main MDP task $\mathcal{M}$. Line 10 splits the state into the specified feature sets for use by the actor network. On line 11 the agent samples an action from the actor using the current state and task. Line 12 executes the chosen action to get a next state s'. Line 13 calculates the auxiliary rewards and main task rewards for this state action pair. The reward, action, and states are then added to the trajectory buffer on line 14. Lines 15 and 16 are book-keeping for the loop. After the episode finishes all samples in the trajectory buffer are sent to the learner on line 18.

4.2.4 Policy Improvement

Similarly to the action-value function defined in the previous section, we can define policy networks as:

$$\begin{aligned}
&\pi_\theta^{\mathcal{T}}(\cdot|s_t) = \mathcal{N}(\mu_{\mathcal{T}}, \mathbf{I}\sigma_i^2) \\
&[\mu_{\mathcal{T}}, \sigma_{\mathcal{T}}^2] = f([\mathbf{e}_1 \odot \mathbf{g}_{1,:}, \mathbf{e}_2 \odot \mathbf{g}_{2,:}, \mathbf{e}_3 \odot \mathbf{g}_{3,:}]; \theta_{\mathcal{T}}), \\
&\mathbf{g} = [g_{\theta_p}(S^\sigma_{\text{proprio}}(t)), g_{\theta_f}(S^\sigma_{\text{features}}(t)), g_{\theta_i}(S^\sigma_{\text{image}}(t))]^T, \\
&\mathbf{e} = [\mathbb{I}^{\pi,\text{proprio}}(\mathcal{T}), \mathbb{I}^{\pi,\text{features}}(\mathcal{T}), \mathbb{I}^{\pi,\text{image}}(\mathcal{T})],
\end{aligned} \tag{4.6}$$

Here $\mathcal{N}(\mu, \mathbf{I}\sigma^2)$ is a multivariate normal distribution with mean μ and a diagonal covariance matrix. Policy parameters are $\theta = \{\theta_{\mathcal{M}}, \theta_{\mathcal{A}_1}, \cdots, \theta_{\mathcal{A}_K}, \theta_p, \theta_f, \theta_i\}$ which are optimized using the reparameterized gradient of $J(\theta)$ based on the learned Q-functions:

$$\nabla_\theta J(\theta) \approx \sum_{\mathcal{T}} \nabla_\theta \mathbb{E}_{a \sim \pi_\theta^{\mathcal{T}}(\cdot|s)} \Big[\hat{Q}_{\mathcal{T}}^{\pi}(s_t, a; \phi) - \alpha \log \pi_\theta^{\mathcal{T}}(a|s_t) \Big], \tag{4.7}$$

The second term is an additional entropy regularization with weight α. Note that different policy representations may use something other than $\mathcal{N}(\mu_{\mathcal{T}}, \mathbf{I}\sigma_i^2)$ without changing how the policy outputs are filtered per task. However, both off-policy learning algorithms in the experiments use a Gaussian policy for computational reasons. Additionally, different algorithms may or may not include the entropy regularization term, or may include it in different parts of the equation.

Figure 4.1 is a diagram of the complete network architecture of a policy network with our applications feature sets. The input layers use the previously described input gating architecture.

Algorithm 3 SAC-X with Multiple State Representations Actor

Input N Number of trajectories
T: Number of steps per episode
ξ: Scheduler period
K: Number of Tasks

function MS-SAC-X ACTOR(N, T, ξ, K)
  for $n \leftarrow 0, n < N$ **do**
    fetch parameters θ
    $\tau \leftarrow \emptyset$ ▷ Initialize new trajectory
    $s \leftarrow$ reset environment
    for $t \leftarrow 0, t < T$ **do**
      if t mod $\xi = 0$ **then** ▷ If scheduler period select new task
        $\mathcal{T}_i \sim Uniform(0, K+1)$
      end if
      $s^{\sigma}_{features}, s^{\sigma}_{proprio}, s^{\sigma}_{image} \leftarrow$ split s into sets
      $a \sim \pi_\theta(\cdot | s^{\sigma}_{features}, s^{\sigma}_{proprio}, s^{\sigma}_{image}, \mathcal{T}_i)$ ▷ Sample new action for task
      $s' \leftarrow$ execute a
      $r \leftarrow [r_{\mathcal{A}_1}(s,a), \cdots, r_{\mathcal{A}_K}(s,a), r_{\mathcal{M}}(s,a)]$
      $\tau \leftarrow \tau \cup \{(s, a, r, s')\}$
      $t \leftarrow t + 1$
      $s \leftarrow s'$
    end for
    send τ to learner
    $n \leftarrow n + 1$
  end for
end function

The outputs are also gated like in the SAC-X paper. For our particular application only proprioceptive and feature groups are enabled for task 0 — thus allowing their inputs to flow into the hidden layers of the f network — while the image network output is zeroed out. Bold lines show which paths in the network are propagated to the output and allow gradients to flow. The red line indicates where the gate zeros the output and as a side-effect prevents gradients from flowing backwards.

In this work we assume that all of the outputs of the g functions are the same shape. These inputs are all summed element-wise. This reduces the amount of zeros flowing into the f network layers, but it requires the assumption that each of the state-spaces can match the same expected internal representations. If this is not true, then outputs of the g network can be passed as separate, unique inputs to the f network. This would allow the f network to specialize to specific g networks in specific tasks, but may slow down the learning by not requiring a common hidden representation between the different feature set combinations.

Algorithm 4 shows the pseudo-code for the learner using the multiple state representation SAC-X approach. We refer to this algorithm as Multi-State SAC-X (MS-SAC-X) Learner. This algorithm runs in a separate process from the Actor in algorithm 3. We only run a single learner

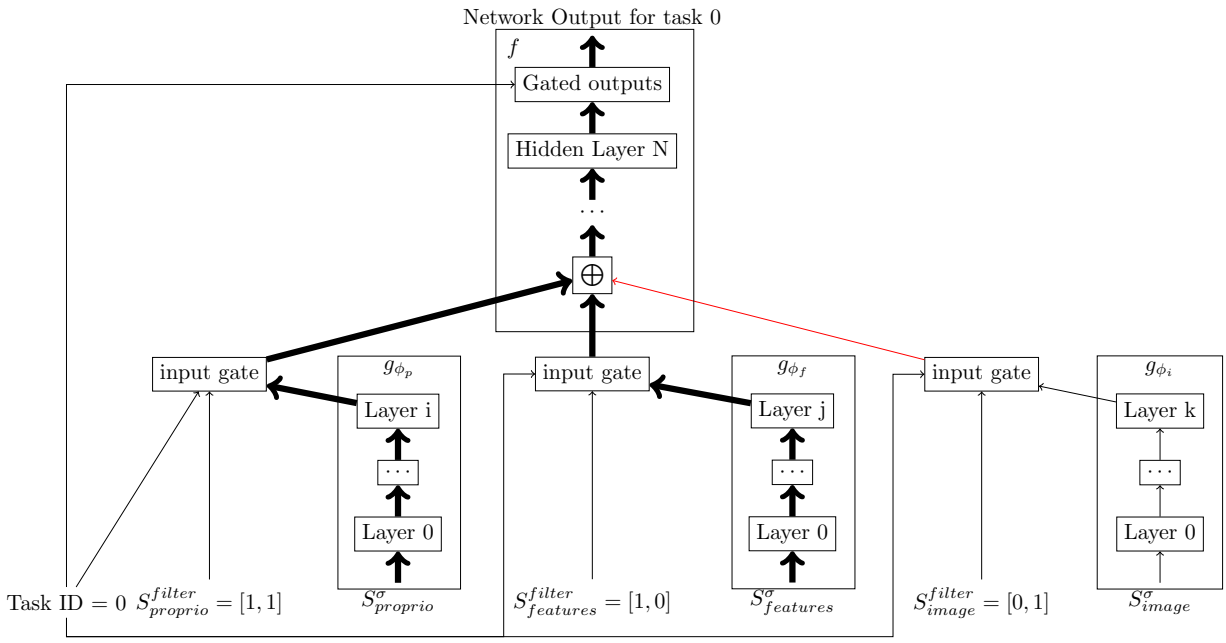


Figure 4.1: Network architecture with specific examples for two tasks evaluated on task 0 and three state sets.

process in our experiments, but in principle you can run multiple and batch gradients from multiple learners before applying the parameter update. Line 2 and 3 initialize the algorithm, setting initial actor and critic parameters and initializing an empty replay buffer B which will be filled by the MS-SAC-X Actor from algorithm 3. Each iteration, a batch b is sampled (shown on line 5). Lines 6 through 12 convert this batch to a set of samples for different randomly selected tasks. On line 8 we sample a random task. Lines 9 and 10 split s_i and s'_i into the feature sets used by the actor and critic. We also extract the specific task reward $r_i[\mathcal{T}_i]$ from the array of rewards r_i. All of these pieces are combined into a tuple and inserted into the modified batch b' on line 11. On line 13 we update the actor and critic using an off-policy learning algorithm. In this thesis we use both SAC and SVG-0 as our off-policy learning algorithms. Line 14 performs book-keeping for the loop. We repeat this whole process until the number of requested parameter updates has been reached.

Algorithm 4 SAC-X with Multiple State Representations Learner

Input N Number of policy updates

function MS-SAC-X LEARNER(N, K)
  $\theta, \phi \leftarrow$ initial actor, critic weights
  $B \leftarrow \emptyset$
  for $n \leftarrow 0, n < N$ **do**
    sample batch b from B
    $b' \leftarrow \emptyset$
    for each sample (s_i, a_i, r_i, s_i') in batch **do**
      $\mathcal{T}_i \sim Uniform(0, K+1)$
      $s^{\sigma}_{i,features}, s^{\sigma}_{i,proprio}, s^{\sigma}_{i,image} \leftarrow$ split s_i
      $s^{\sigma'}_{i,features}, s^{\sigma'}_{i,proprio}, s^{\sigma'}_{i,image} \leftarrow$ split s_i'
      $b' \leftarrow b' \cup \left\{ \left(s^{\sigma}_{i,features}, s^{\sigma}_{i,proprio}, s^{\sigma}_{i,image}, a_i, r_i \left[\mathcal{T}_i\right], s^{\sigma'}_{i,features}, s^{\sigma'}_{i,proprio}, s^{\sigma'}_{i,image}, \mathcal{T}_i \right) \right\}$
    end for
    update policy parameters θ and ϕ using batch b'
    $n \leftarrow n + 1$
  end for
end function

4.2.5 Asymmetric Actor-Critic with Different State Spaces

Our approach shares similarities wit the existing asymmetric actor-critic technique from Pinto et al. [74]. The asymmetric actor-critic allows the state space of the critic to be enhanced with additional information that may not be available at test time. This can simplify the problem of credit-assignment, leading to faster convergence of the critic network. After training, the critic is not necessary, and can be discarded, similar to how our auxiliary tasks are discarded after training. In this manner, our approach is an extension of the asymmetric actor-critic method to a multi-task setting.

Additionally, the asymmetric actor-critic technique is complementary to our approach. For example, we can restrict the critic state-filters ($S^{filter}_{Q,i}$) for all tasks to use only the feature observations. The goal being that it is easier to learn credit assignment in the distilled feature space, which will lead to faster critic convergence, and therefore better actor updates. Additionally, we have the flexibility to have different asymmetric representations for each task by having different filters for each critic task. We refer to this as the asymmetric actor-critic setting in our empirical evaluations.

4.2.6 Designing Auxiliary Tasks

In the previous sections, we described how to do learning with auxiliary tasks that use multiple state representations. However, we did not give specific details on how one can design these tasks when attempting to solve a particular problem. Unfortunately, it is not possible to say up-front which auxiliary tasks are useful and which are necessary to successfully learn the goal task, but

in this section, we give some general ideas of how to start.

The auxiliary tasks can be split into four major groups:

1. The task is required at test time
2. The task provides a shaped reward for a test time task
3. The task has relaxed constraints compared to the test time task goal
4. The task encourages exploration to potentially useful parts of the state-action space

Test Time Tasks In our work, we are considering only a single test time task, but in general you may have more than one. Additionally, we are considering only a sparse reward version in a specific state space. However, depending on the task one may want to use a more complicated reward function to bias behavior in a particular manner or have test time tasks with different state spaces. Before designing any helper auxiliary tasks, one must first determine what the test time tasks reward and state spaces will be.

Reward Shaped Tasks Shaped rewards for test tasks, have been used for a long time in Reinforcement Learning [70]. While such rewards can greatly speed-up learning, it is very easy to unintentionally learn useless policies that exploit the shaping function rather than achieve the desired objective. By adding auxiliary tasks with shaping, yet keeping a task with the original reward function for use at test time, one can get the benefits of shaping without the potential downsides. Even if the shaped auxiliary task learns an exploitative policy, it will still generate useful training data for the test-time sparse reward task. Therefore, adding various shaped reward auxiliary tasks is a good starting point for any problem. Also, because multiple auxiliary tasks can be created, one can try different shaping functions that emphasize different aspects of the desired behavior.

Relaxed Constraint Tasks Achieving all conditions necessary for a sparse-reward goal-oriented task can be quite difficult to achieve. Relaxed constraint tasks can help guide the agent to completing the full task by incentivizing it to complete easier versions of the task. For example, if the test time task is to drive a robot to a specific (x, y) location. We could potentially create two relaxed constraint tasks for this goal task. We could have a task that gives reward for matching the x location only and a task that gives reward for matching y location only. Because matching only part of the full test-time task goal is easier to achieve, the agent will likely see more rewards, earlier and be able to learn a policy to achieve these exploration goals earlier. Additionally, once the agent is matching one of the coordinates, random exploration on-top of this single-coordinate matching policy is more likely to hit the true goal conditions. Another way of relaxing the constraints is to increase the tolerance of the sparse-reward goal task. For example, instead of needing to be within 10cm of the goal location, we can have tasks that give rewards when within 20cm and 100cm of the goal location. Similarly, once in this region, the agent is closer to the goal, and so random exploration from this area is more likely to produce examples of positive reward in the sparse goal task.

Exploration Tasks Tasks encouraging exploration generally focus on rewarding reaching new parts of the state-action space. Unlike the shaped reward tasks and the relaxed constraint tasks, the goal of exploration tasks is not to complete the test time task goal or parts of the test time task goal. The goal of these tasks is simply to explore and get "interesting" data for the policy update to utilize when training the other tasks.

Given the above guidelines, it is easy to invent many different auxiliary tasks. Start with the test time tasks, then add shaped and relaxed constraint versions of these tasks. Additionally, include some exploration focused tasks to ensure good coverage of the state-action space. Given the large number of tasks one can generate this leads to two questions:

1. Can we add too many tasks?
2. What if we add detrimental tasks?

Number of Tasks Given the above guidelines, it is generally easy to invent many different auxiliary tasks. The concern then becomes how many tasks are too many tasks. We argue this largely depends on the scheduler. In this thesis we use a uniform random scheduler. As the task set grows, this will lead to each task having less time scheduled on average. While all tasks collect training data, if the scheduling period is too small, the task switching can interfere with the completion of any one task. Previous work used a more intelligent scheduler that attempted to learn a good scheduling function based on the returns from the auxiliary tasks [80]. More research into smarter schedulers can definitely improve the scalability regarding how many tasks it is feasible to learn with. However, despite this, in our Ball-in-a-Cup experiments, we use 10 or more tasks in various experiments and did not see any negative effects. The general rule of thumb should be to add lots of auxiliary tasks, but tune the scheduler period so that the task interference does not become too large.

Detrimental Tasks Finally, a remaining concern is whether adding a detrimental task can cause the policy learning to fail. For example, a task with a reward that is the inverse of the test-time task. Such a task would still generate rewards that would help train the test-time policy to avoid the state-actions demonstrated by this inverse task policy. The downside is that after scheduling the inverse task and then scheduling a non-detrimental task the agent will be starting far from the goal. Another type of detrimental task is one that incentivizes the agent to do nothing. Unlike the inverse policy task, the data collected by this task would likely be useless for the test-time policy. However, given that it is only scheduled randomly, it is unlikely to cause major issues. Later in the Ball-in-a-Cup experiments section we present results where such a task was added, and yet the policy learning still converged. Overall, the rule of thumb should be to try to avoid detrimental tasks, but as long as the number of detrimental tasks is small, it will likely not break the policy learning.

4.3 Empirical Results from Simulated RoboCup SSL Domains

We now apply our technique to simulated tasks from the RoboCup SSL domain. During training we assume access to both features available from the vision detection system such as robot

positions and estimated velocities. At test time our policy receives only images as state input and must extract position and velocity information from these images. We demonstrate that our technique works and that we can learn policies based on the vision data faster than if we had learned with only vision data from the start.

4.3.1 Environment Description

We train policies for a simple navigation task. The goal is for a single robot on a 12m x 9m field to navigate to a fixed position and orientation with some allowed amount of error in both. This is similar to the `go-to-ball` skill learned in chapter 2, however, here we are not concerned with a ball but simply controlling position and orientation. Additionally, in this environment we will use a fixed goal position of the center of the field with the orientation of the robot facing right. Figure 4.2 shows an example of the robot in the correct goal position and orientation. Finally, while we will be using some tasks with features and reward shaping similar to those used in chapter 2, ultimately we will be trying to learn a policy that works using only simulated images and a sparse reward. The agent controls the robot at 10Hz for 300 steps (about 30 seconds). Each episode starts with the robot at a uniform random position and orientation in the field.

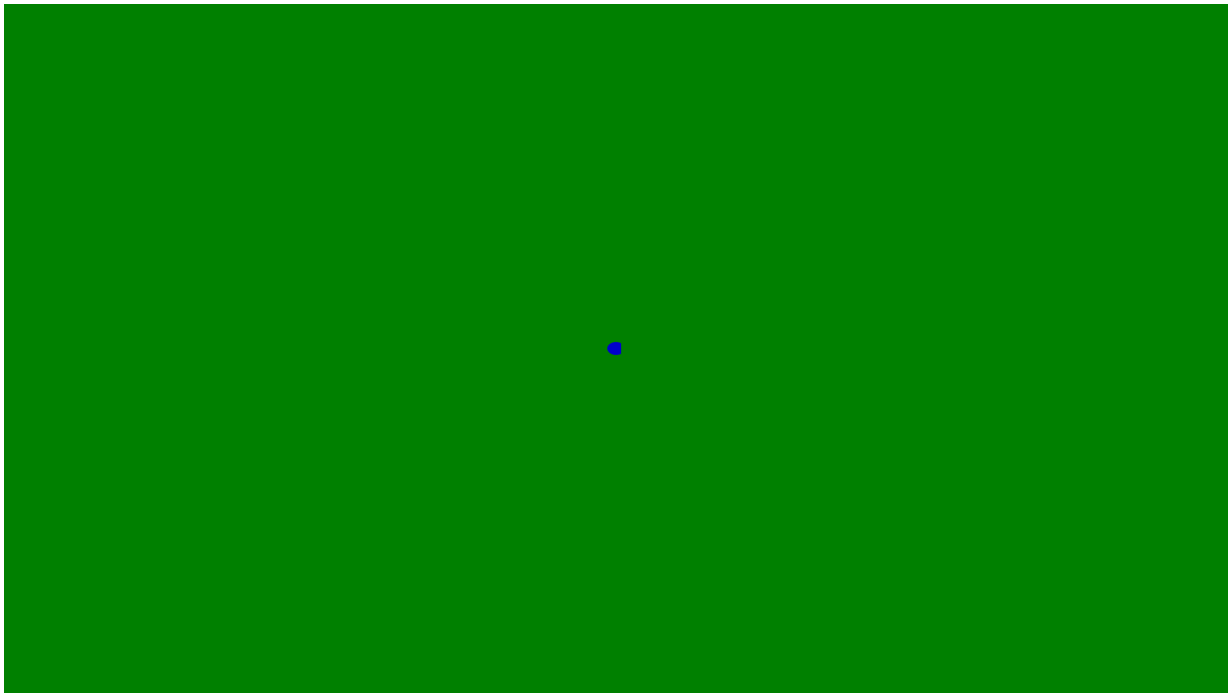


Figure 4.2: An image of the simulated robot in the desired goal configuration

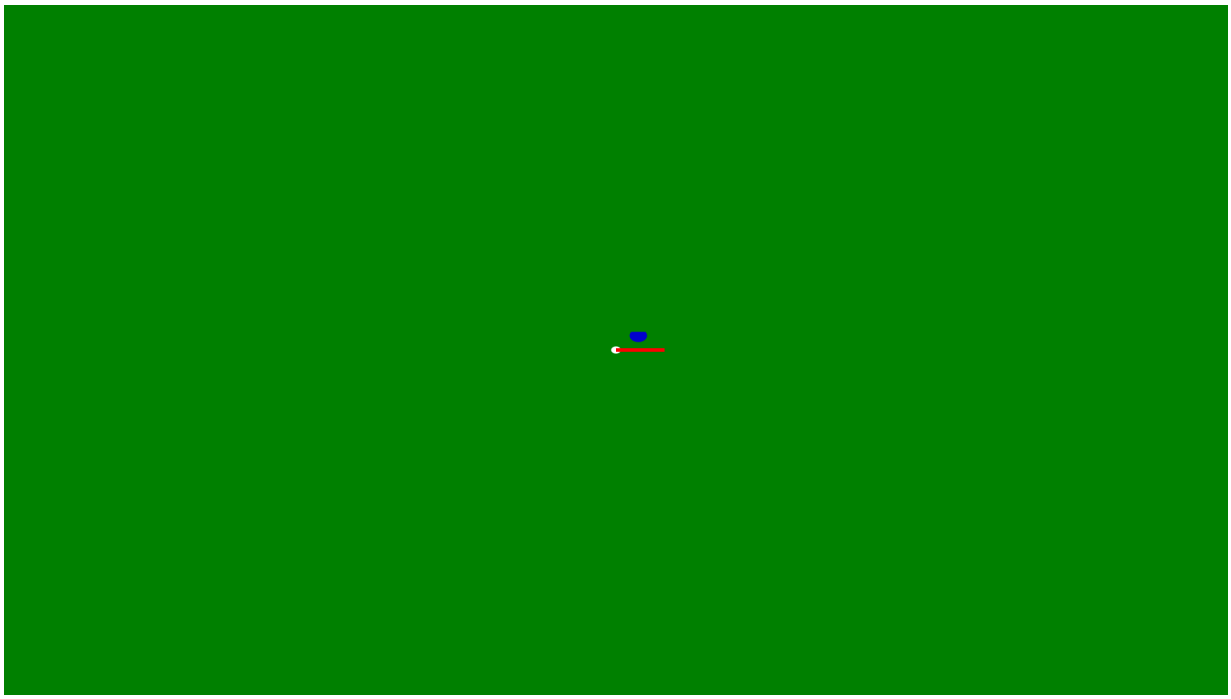
Table 4.1 presents the observation groups, along with the sizes that are used in the simulated SSL navigation task. We use the position and velocity features that would be available on the real SSL robots using the vision system. We construct the three image types by rendering a high resolution overhead view of the robot and field, which corresponds to the high resolution overhead camera view processed by the vision system. These images are cropped at various

locations and down-scaled to get the images used as the state in the test-time task. We use three image types: field image, robot image, goal image. The field image provides a low-res view of the overall field, which has enough resolution to approximately determine robot position. The robot image is created by cropping the high resolution overhead image around the robot's current position and then down-scaling. This provides the necessary resolution to determine the robot's current orientation. Finally, the goal image is a cropped area around the goal position that is down-scaled. This allows the agent to fine-tune its position when it is near the goal so that it can enter the sparse reward region. A history of the current images and the previous two time-steps are stacked to allow the agent to estimate linear and angular velocity. In this domain the proprioceptive features group $S^{\sigma}_{proprio}$ is not needed.

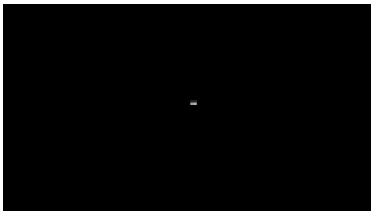
State-group	**Observation**	**Shape**
$S^{\sigma}_{\text{features}}$		
	Robot Position	2
	Robot Orientation (sin/cos of angle)	2
	Robot Linear Velocity	2
	Robot Angular Velocity	1
$S^{\sigma}_{\text{proprio}}$		
	Unused	—
$S^{\sigma}_{\text{images}}$		
	Stacked Field Image	$112 \times 84 \times 1 \times 3$
	Stacked Robot Image	$32 \times 32 \times 1 \times 3$
	Stacked Goal Image	$32 \times 32 \times 1 \times 3$

Table 4.1: State group definitions with observation sources and shapes for the simulated SSL navigation task.

Figure 4.3a shows an example of the full resolution overhead image that is cropped and down-scaled to create the three image inputs. The goal region is shown as a white circle at the center of the field. The center of the robot must be within that circle to get a +1 reward. The red line shows the desired orientation. The flat part of the robot must be facing the right and within 5 degrees of being perpendicular to that line to get the sparse reward. Figure 4.3b shows the down-scaled field image from figure 4.3a provided to the agent. Note that the robot takes up approximately 1 pixel in this image, meaning that it is enough to estimate coarse position but no orientation. And the position information is not fine-grained enough to get within the circle consistently. Figure 4.3c shows the cropped and down-scaled image of the goal region provided to the agent. In this case, only part of the robot is within the cropped region. So we see the rounded back of the robot in the upper right corner of this image. Figure 4.3d shows the cropped and down-scaled image of the robot provided to the agent. This image is always cropped to the robot's current position, so we can see the full robot in its current orientation. When the robot is at the goal, the goal image and the robot image will be very similar, just the goal image will be cropped to a bigger region.

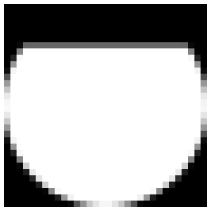


(a) An example of the simulated SSL field and robot for the navigation task. This is the full resolution that is split into multiple images and downsized for processing by the actor and critic network. The white circle shows the sparse goal region. The red line shows the goal orientation.



(b) Example of downsized full-field image provided to the agent for the simulated SSL navigation task.

(c) Example of downsized goal-region image provided to the agent for the simulated SSL navigation task.



(d) Example of downsized robot image provided to the agent for the simulated SSL navigation task.

Figure 4.3: Examples of image observations given to agent during simulated SSL navigation task

4.3.2 Auxiliary Task Description

We use two different reward functions when training. Table 4.2 shows these two reward functions.

Reward ID	Reward Name	Reward Function
1	Sparse Navigation	+1 if robot is within 10cm of the goal position and 5 degrees of the goal orientation.
2	Gaussian Shaping	See equation 4.8

Table 4.2: Reward functions used for different tasks. Reward 1 is the main task reward that we use at test time.

Equation 4.8 shows the shaped reward equation.

$$r_2 = \frac{5}{\sqrt{2\pi}} \exp -\frac{x^2 + y^2 + \theta^2}{2} - 2 \tag{4.8}$$

In equation 4.8 x and y are the position coordinates of the robot with respect to the center of the field and θ is the orientation of the robot with respect to the x-axis of the field. We reuse the same coordinate frames as those shown in chapter 2 figure 2.3.

We use two different state groups: $S^{\sigma}_{features}$ and S^{σ}_{images} as shown in table 4.1. We use two different task state-spaces: features and images.

Tasks are a combination of state space type and reward function. We refer to specific combinations by a reward ID, followed by either an "F" for the feature state-space $S^{\sigma}_{features}$ or "P" for the pixel state-space S^{σ}_{images}. For example, 1P rewards to the sparse task reward with the image state-space, which is the task we are interested in at test time. We use the following tasks in our experiments: 1F, 2F, 1P, 2P.

4.3.3 Comparison of State Space Combinations

We trained three different agents: a features agent, a pixels agent, and a features+pixels agent. The latter being the technique contributed by this thesis. For a fair comparison both the pixels only and features+pixels agent both use the asymmetric actor-critic technique. Therefore any speed-ups can be attributed solely to the contribution of learning with multiple state representations. Table 4.3 shows the tasks used during training for each agent as well as the task we are interested in at test time. All of the agents use the same network structure, but with different state groups enabled/disabled based on train tasks. Additionally, all agents are trained with the Soft Actor-Critic (SAC) algorithm. Appendix A.4 provides details on the specific network structures and hyperparameters used.

We train each agent 3 times from scratch to show consistent behavior and that the results are not very dependent on the randomly chosen initial parameters or the random exploration noise. For each run, we save the parameters every 100 updates and then evaluate the test time policy deterministically 10 times. We sum the rewards for each of these test runs and then average

Agent	Train Tasks	Test Tasks
features	1F,2F	1F
pixels	1P,2P	1P
features+pixels	1F,2F,1P,2P	1P

Table 4.3: The combinations of train and test tasks used for each agent.

across the 3 independent runs. The lines show average performance and the shaded regions show minimum and maximum performance across all runs.

Figure 4.4 shows the results of the evaluation. The solid lines show the mean across the runs and the shaded area shows the minimum and maximum performance for each curve. Firstly, we see that regardless of the training tasks and the test time state-space all three agents are able to learn a similarly performing policy. As expected learning only with pixel state-space tasks converges the slowest, even using the asymmetric actor critic technique. It reaches similar performance after about 60,000 steps and then fully stabilizes at around 70,000. The training continued for more steps than displayed and remained stable, but the plot has been clipped for easier comparison between methods.

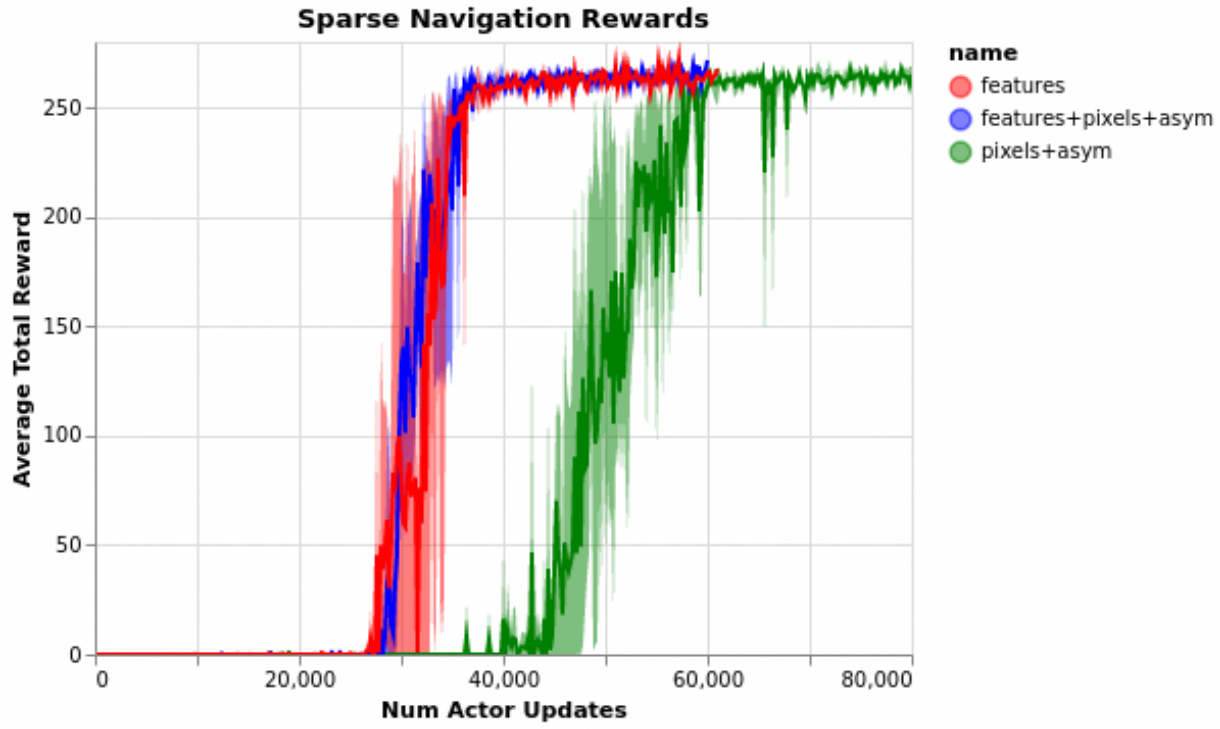


Figure 4.4: Learning curves comparing performance of learning only with features, learning with pixels and features and learning only with pixels.

More interestingly, we see that the thesis approach of learning with both pixel and feature state space tasks simultaneously and the baseline of learning with just feature based tasks perform

similarly. The mean of the baseline features only runs and the features+pixels+asym runs are approximately the same. Based on this result, for this domain, there does not appear to be a difference in speed, showing that the thesis approach is capable of learning a sparse pixel state space task about as fast as just learning a sparse feature state space task. While the means are approximately the same, there does appear to be more variance in the features only baseline. With the shaded region showing that the maximum performance of the features only is slightly better than the maximum of the features+pixels+asym. For this simple task, the two techniques perform almost the same. However, as will be shown in later sections, this is not always the case. For more complex tasks and more realistic images the gap between the features only baseline and the thesis technique will increase

One may also wonder why the agents sometimes drop in performance for a small span and then quickly improve performance again. Looking at the executions of these policies visually, we see that when the performance drops, it is not because the agent has completely forgotten how to do the task. Instead, the agent still drives near to the goal region, but isn't quite inside it. It may get small amounts of reward from passing through the goal region though. Once the agent does a few more policy updates it fine-tunes its behavior to more accurately drive to the goal region, causing the performance to jump back up.

4.3.4 Comparison Between Tasks

Figure 4.5 shows a comparison of each of the training tasks from the features+pixels agent. Two of the tasks (2P, 2F) use shaped rewards, which have a maximum total episode reward of 0. Two of the tasks (1P, 1F) use sparse rewards, which have a maximum total episode reward of 1. Note that in practice the maximum is not achievable unless the agent starts at the goal and remains there for the whole episode. The hypothesis of our approach is that the the feature tasks with shaped rewards converge first, which generate good training data and a good internal representation for the high-dimensional pixel state space. Once the shaped rewards are converged for all the state spaces we should see the sparse rewards for the different state spaces converge. This is exactly what we see in the figure. At around 4,000 actor updates the 2F task (i.e. shaped features) converges. Shortly after at about 5,000 updates the 2P task (i.e. shaped pixels) converges. After convergence of these shaped tasks the agents begin getting lots of examples of hitting the sparse reward region and at about 30,000 actor updates the remaining two tasks using the sparse rewards converge.

4.3.5 Qualitative Performance of Sparse Pixels Policy

Figure 4.6 shows an example of an execution of the trained sparse pixel policy (i.e. 1P). The left column shows the color simulated images with the goal location and orientation drawn on top. The images are cropped to allow for easier viewing in the document. The goal location is the white circle and the goal orientation is marked by the red line. The right column top image shows a cropped version of the full field image given to the agent. The bottom left image in the right column is the goal region image given to the agent. When the robot is at the correct location the robot should appear in this image at approximately the center.The bottom right image in the right column is the robot image given to the agent which provides orientation information. Overall,

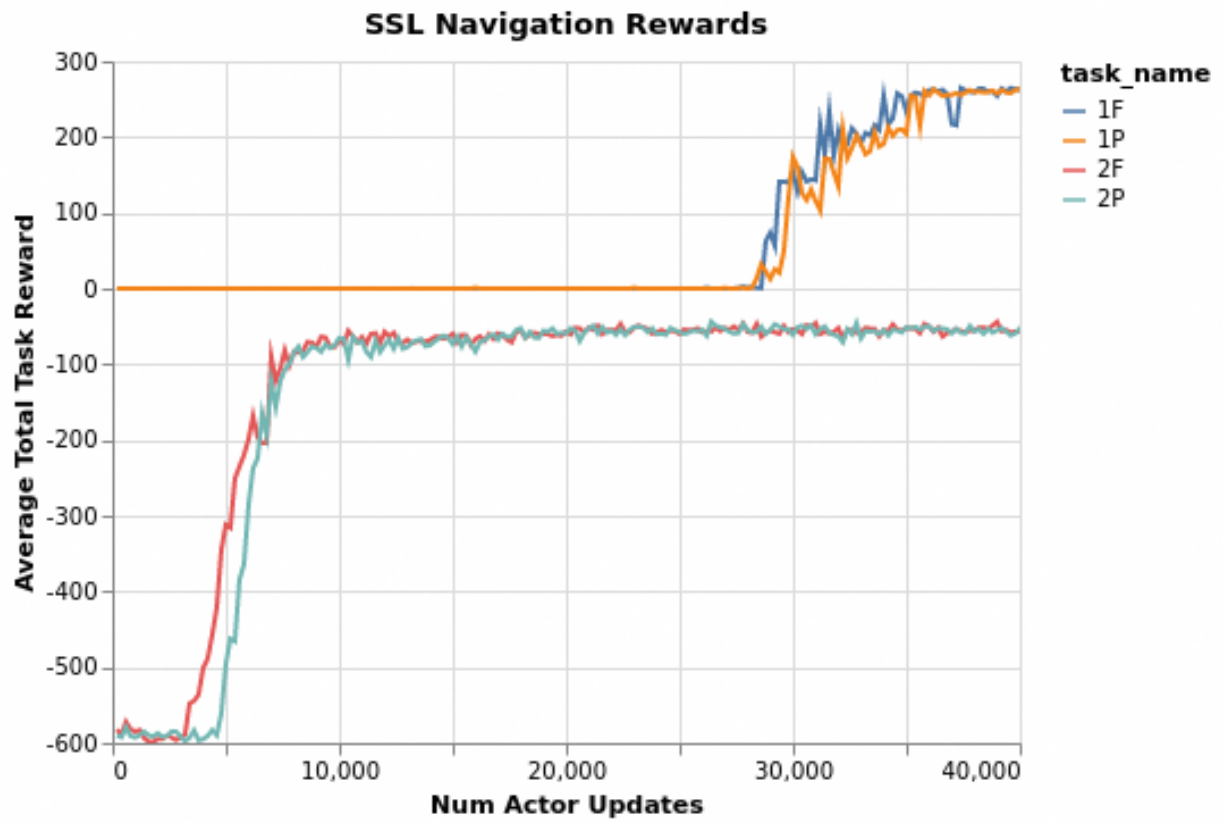
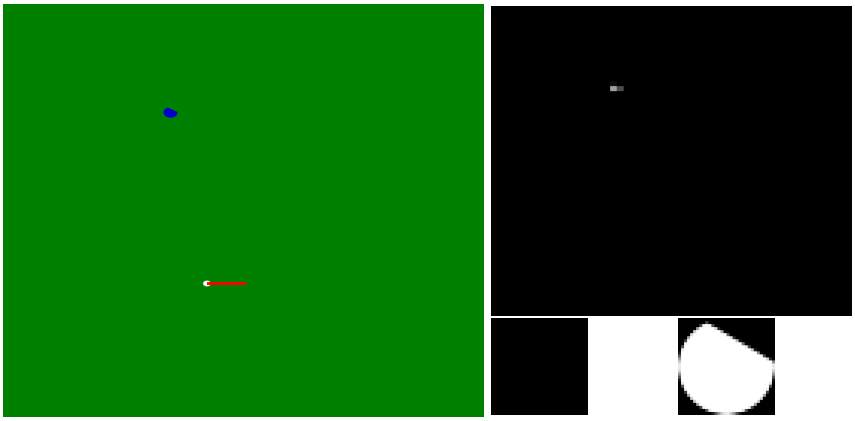


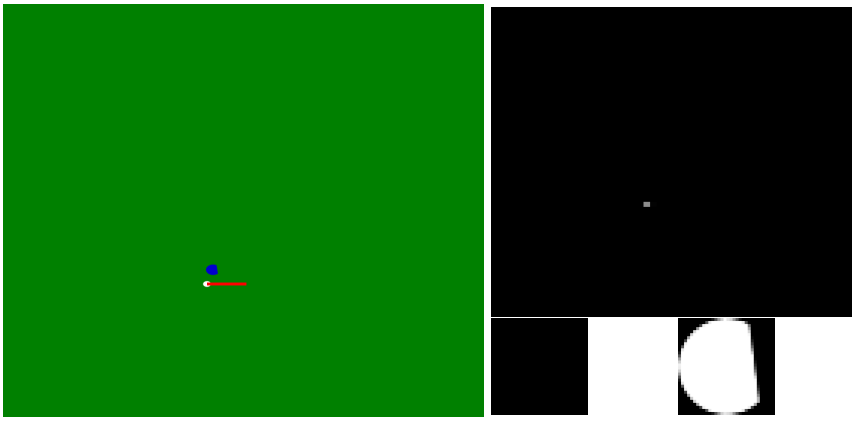
Figure 4.5: Learning curves comparing the performance of each train task from the features+pixels+asym agent.

across most runs, we see the robot twist towards the goal orientation while simultaneously moving towards the goal location. Often it has reached the goal orientation log before it has gotten to the goal location.

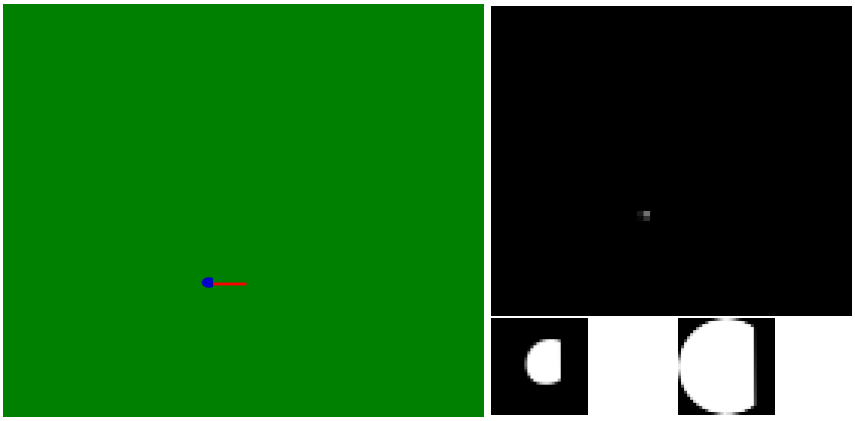
Figure 4.6a shows the initial state in this episode. The robot starts facing up and above the goal location. As the robot is far from the goal region the goal image is empty. You can see that the field image given to the agent gives a rough position and the robot image gives a good estimate of current orientation. Figure 4.6b shows the robot after about 20 time steps. The robot has twisted to be close to the goal orientation and has moved much closer to the marked goal location. Finally, figure 4.6c shows the robot after it has reached the goal location and orientation after about 30 time steps. We can see the goal image now contains the robot and the robot is at approximately the right location. This goal image allows the agent to fine-tune its position once it is near the goal.



(a)



(b)



(c)

Figure 4.6: A sequence of images from an execution of the trained sparse pixel policy on the SSL navigation task.

4.4 Empirical Results from Ball-in-a-Cup Task

The previous section showed that our technique works for our running example of RoboCup SSL. In this section we show that this technique works on the Ball-in-a-Cup task preformed by a robot arm. This demonstrates that the contributed technique works outside of the RoboCup domain and on a more complex domain. Additionally, we demonstrate this technique directly on real-robot hardware. Showing that the speed-ups are enough to learn from scratch, on real-robot hardware, a policy for a dynamic task that utilizes only raw images and proprioceptive features.

4.4.1 Ball-in-a-Cup Environment

This section describes the experimental setup for both simulation and the real-robot. Table 4.4 shows the observations in each previously defined state-group (i.e. $S^{\sigma}_{\text{proprio}}$, $S^{\sigma}_{\text{features}}$, $S^{\sigma}_{textimage}$) along with the observation source and shape.

State-group	Data Source	Observation	Shape
$S^{\sigma}_{\text{features}}$			
	Vicon		
		Cup Position	3
		Cup Orientation (Quaternion)	4
		Ball Position	3
	Finite Differences		
		Cup Linear Velocity	3
		Cup Angular Velocity (Euler Angles)	3
		Ball Linear Velocity	3
$S^{\sigma}_{\text{proprio}}$			
	Sawyer		
		Joint Position	7
		Joint Velocities	7
	Task		
		Previous Action	4
		Action Filter State	4
$S^{\sigma}_{\text{images}}$			
		Stacked Color Front Image	$84 \times 84 \times 3 \times 3$
		Stacked Color Side Image	$84 \times 84 \times 3 \times 3$

Table 4.4: State group definitions with observation sources and shapes.

We use a Sawyer robot arm from Rethink Robotics [3], which has 7 Degrees of Freedom (DoF). The policies control velocities of 4 out of the 7 DoFs (joints J0, J1, J5, J6), which are sufficient

[3] `http://mfg.rethinkrobotics.com/intera/Sawyer_Hardware`

to solve the task. Unused DoFs are commanded to remain at a fixed position throughout the episode.

For the real robot, position and orientation features are determined by an external Vicon motion capture system. IR reflective markers are placed on the cup and ball and the Vicon tracks clusters to determine positions and orientations. Estimates are accurate to sub-millimeter precision, have low latency, and low noise.

Our setup includes two external cameras positioned roughly orthogonally to each other (see Figure 4.7). RGB frames of 1920×1080 pixels are captured at 20 Hz and down-sampled via linear interpolation to 84×84 pixels. A stack of the current frame and 2 previous frames are passed into the neural network. Figure 4.8 shows pictures of the real camera images seen by the robot during training and testing. No effort was made to finely calibrate camera positions, crop images, tune white balance or correct for lens distortions. Instead, we rely on the adaptability of the learning system to cope with these negative visual effects.

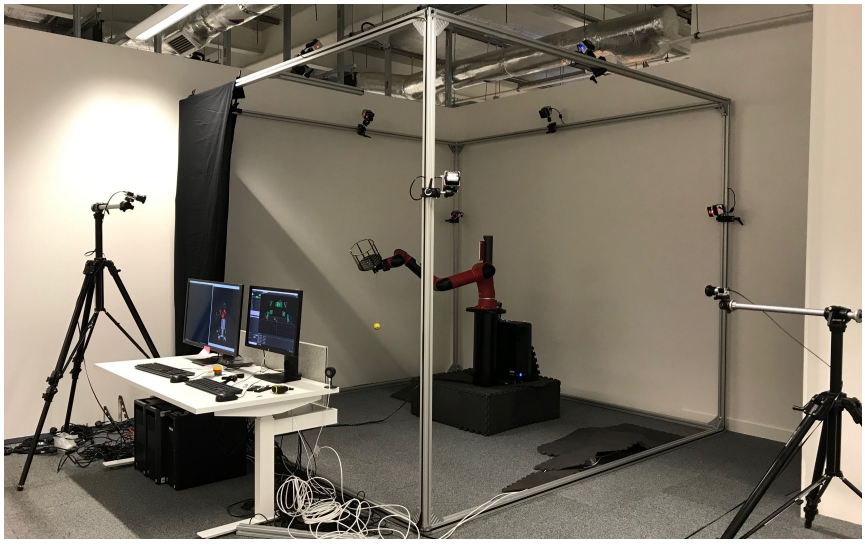


Figure 4.7: Picture of the real robot setup.

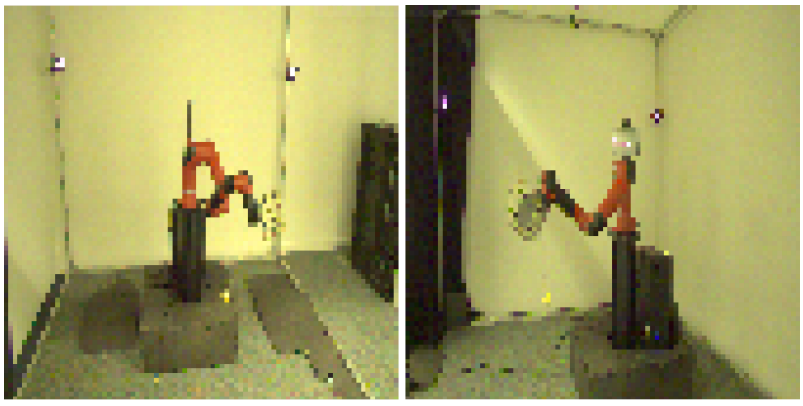


Figure 4.8: Agent's perspective: down-sampled front camera (left) and down-sampled side camera (right) with (84×84) pixels.

Data from proprioception (100Hz), Vicon ($\sim$100Hz) and cameras (20Hz) are received asynchronously. State observations are sampled at 20Hz with no buffering (i.e. latest data is always used). The agent controls the robot synchronously with action sampling at 20 Hz. The control rate was chosen to allow for dynamic reactions to the current ball position and velocity, while not being so high that the learning algorithm is overwhelmed.

Physical Setup

The ball has a diameter of 5cm and is made of foam to prevent damage to the robot while swinging. A Kevlar string of 40 cm is used to attach the ball to a bearing fixed to the robot's wrist. The bearing's axis of rotation is coincident with the robot's wrist axis of rotation to prevent winding of the string while swinging the ball — when the ball goes around the wrist the string rotates freely via the bearing.

The cup is 3D printed, and a woven net is attached to it ensuring visibility of the ball even inside the cup. The cup is attached to the wrist after the bearing. Experiments are performed with two cup sizes: a large cup with 20cm diameter at 16cm height and a small cup with 13 cm diameter and 17cm height. Unless otherwise stated, experimental results are for the large cup.

Robot Safety

Training with the real robot presents safety issues for the robot and its environment. An agent sending a large, random velocity action to the robot may cause self-collisions or collisions with the environment. Additionally, quickly switching between minimum and maximum velocities incurs wear to the robot's motors. We utilize an action filter as well as a number of hard-coded safety checks to ensure safe operation.

The Sawyer has built in self-collision avoidance. To prevent collisions with the workspace, and keep everything within the trackable volume, we limit the joint positions to within a safe region.

To prevent high frequency oscillations in velocity commands we pass the network output through a low-pass first order linear filter with cutoff frequency of 0.5 Hz. The cutoff frequency was chosen qualitatively so that the robot can build up large velocities before hitting joint limits, while still preventing switches between minimum and maximum velocities. This filter has an internal state which depends on the history of actions. To keep the problem Markovian, we provide the agent with an internal state of the action filter as part of the proprioceptive observations.

Episode Resets

Training is episodic with a finite horizon of 500 steps per episode, resulting in a wall clock time of 25 seconds per episode. Intention policies are randomly switched every 100 steps (dividing each episode into 5 subsets). This increase diversity in starting positions for each policy. Episodic training in simulation is easy, as the simulator can be reinitialized each episode. On the real-robot, resetting is significantly more challenging due to string wrapping and tangling around the robot and the cup. The string is not easily modelable or observable, making it difficult to come up with a principled method of planning an episode reset. Instead, a simple, but effective means is utilized to allow for automatic resets, thus allowing for unattended training. The robot

commands the joints to drive to positions from a predefined list at random. Positions were chosen to maximize the twisting the arm undergoes when moving between positions. This procedure is run until it is detected that the string is untangled.

Simulation

We use the MuJoCo physics engine for simulation experiments. A model of the robot's kinematics and dynamics was implemented using the parameters provided by the manufacturer. We do not attempt to perfectly mimic the particulars of the real robot (i.e. via system identification). The simulation is mainly used as a tool to gauge asymptotic learning behavior, our real-robot experiments make no use of the simulation. Observations come directly from computed simulation state, meaning that the observations are noise free. This is particularly important for the image observations as they are rendered with a plain background and do not contain real camera artifacts like distortion or motion blur. Figure 4.9 shows the simulation setup along with samples of the simulated camera images.

Figure 4.9: Example of the simulation (left column) along with what the agent sees when using the images (right column).

4.4.2 Experiments

Task Descriptions

Table 4.5 shows the reward functions used in the simulation and real-robot experiments. Ball position is always w.r.t. the cup coordinate frame. Note that r_5 is the main sparse reward function for the Ball-in-a-Cup task (i.e. the reward we care about at test time).

Reward ID	Reward Name	Reward Function
1	Ball Above Cup Base	+1 if ball height is above the base of cup in the cup frame
2	Ball Above Cup Rim	+1 if ball height is above rim of cup in cup frame
3	Ball Near Max Height	+1 if ball height is near the maximum possible height above cup in cup frame
4	Ball Near Opening	Shaped distance of ball to center of cup opening
5	Sparse Catch	+1 if ball in cup
6	Shaped Ball Above Cup Base	See equation 4.9
7	Shaped Swing up	See equation 4.10
8	Do nothing (distractor task)	Negative reward proportional to joint velocities

Table 4.5: Reward functions used for the different tasks. Reward 5 is the main reward for the actual Ball-in-a-Cup task.

Figure 4.10 illustrates the different cup reference frames and positions used for reward computation (e.g. the base, rim, etc.). The formula for r_6 is defined in equation 4.9, which was tuned to return $[0, 1]$ based on minimum and maximum ball height respectively. The reward shaping of r_7 uses a 2D Gaussian with $\mu = 0$ (cup base center) and $\sigma = 0.09$ for both X and Y axes, as in equation 4.10. In this case, σ is defined so that values within $[-2\sigma, 2\sigma]$ coincide with the cup diameter and return zero when the ball is below the cup base.

$$r_6 = \frac{1 + \tanh(7.5z)}{2}, \tag{4.9}$$

$$r_7 = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}}. \tag{4.10}$$

As discussed in previous sections, we use three state-groups: $S^{\sigma}_{\text{proprio}}$, $S^{\sigma}_{\text{features}}$, $S^{\sigma}_{\text{images}}$. We use two main state-spaces for the different training tasks, which we refer to as feature state-space and pixel state-space. Feature state-space tasks have both $S^{\sigma}_{\text{proprio}}$ and $S^{\sigma}_{\text{features}}$ enabled. Pixel state-space tasks have both $S^{\sigma}_{\text{proprio}}$ and $S^{\sigma}_{\text{image}}$ enabled.

Tasks are a combination of state space type and reward function. We refer to specific combinations by a reward ID, followed by either "F" for feature state-space or "P" for pixel state-space.

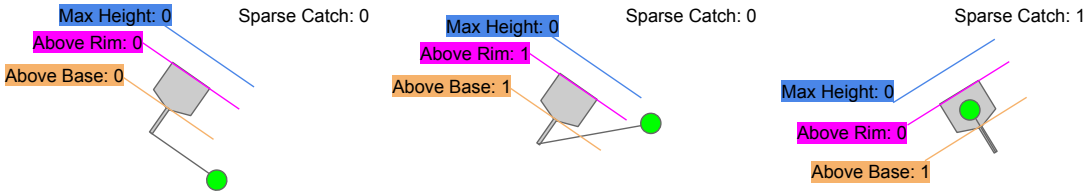


Figure 4.10: Parts of the cup referred to in the task reward function definitions.

For example, 1F refers to task with reward 1 and feature state-space.

Simulation Results

We performed an ablation study in simulation, so that training runs can be parallelized and meaningful statistics can be easily gathered. To evaluate we used the following procedure: after each episode of training we freeze the weights of the main task policy and run an evaluation episode for the sparse catch task (using the appropriate state-space) and report the cumulative reward.

Figure 4.11 shows the results of the ablation experiment. Each curve in the plot is the mean of 10 independent runs of the average reward of the evaluated task. The yellow curve shows an evaluation of task 5F when trained with tasks 1F, 2F, 3F, 4F and 5F. As hypothesized, training with only features converges fastest — presumably because it is easier to learn with feature state-space given the low dimensionality and expressiveness. With an episode time of 20 seconds, learning succeeds in time equivalent to approximately 5.5 hours of real-time training. The purple curve shows the evaluation of task 5P when trained with tasks 1P, 2P, 3P, 4P, and 5P. Using only images results in training times that are approximately 8 times slower. This corresponds to an estimated 1.6 days of continuous training time on a real robotic system. In practice, this time would be even higher due to episode resets and the addition of noise in the observations.

The red and green curves show the results of our method. The red curve depicts the evaluation of task 5P when training is performed with tasks 1F, 2F, 3F, 4F, 5F, 1P, 2P, 3P, 4P and 5P, with $\forall i, S_{Q,i}^{filter} = S_{\pi,i}^{filter}$. This leads to a significant speed-up in learning time compared to training with images alone. Our method is slower by only a factor of 2 compared to training from only features. The green curve shows the evaluation of task 5P when trained with tasks 1F, 2F, 3F, 4F, 5F, 1P, 2P, 3P, 4P and 5P, but this time with the asymmetric actor critic technique (i.e. $S_{\text{features}}^{\sigma}$ enabled in all critic filter lists and $S_{\text{image}}^{\sigma}$ disabled in all critic filter lists). This leads to a slight improvement in early training, likely due to a speed-up in convergence of the critic network.

Figure 4.12 shows a series of frames from a typical successful run of the sparse reward pixel policy in simulation. Each frame has four quadrants. The left quadrants are the full resolution images generated by the simulated cameras. These are down-scaled to produce the low resolution images in the right quadrant which are passed to the agent. Figure 4.12a shows the start of the episode with the ball hanging statically below the cup and the arm in the standard starting position. Figure 4.12b shows the initial swing of the arm to the right. Figure 4.12c shows the arm reversing direction with the correct timing so that the balls momentum increases. Figure 4.12d shows the arm continuing small swings to finish building the ball's momentum. Figure 4.12e shows the ball just before the catch. Pay attention to the upper row where it is clear the ball is not yet in the cup. If you look at the bottom row it is not clear if the yellow ball is in the cup or

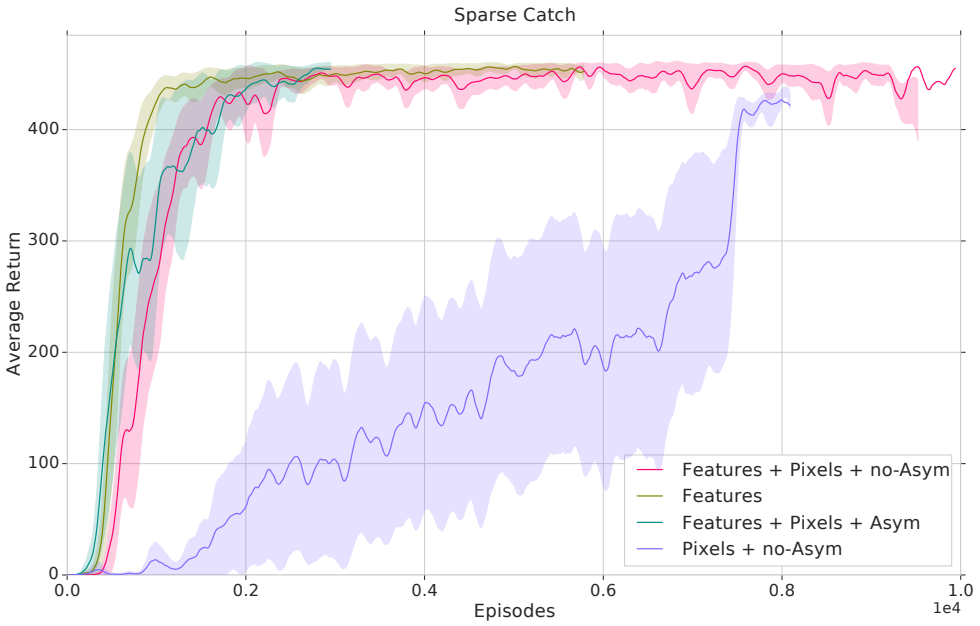


Figure 4.11: Comparison of simultaneously learning in different state spaces vs baselines in simulation.

not. This highlights the fact that both cameras are necessary for this task. Finally figure 4.12f shows the arm just after successfully catching the ball. The arm remains mostly motionless for the remainder of the episode, making only slight movements to the left.

Figure 4.12: Sequence of frames taken from a run of the learned simulated Ball-in-a-Cup policy using images.

Learning from Features on a Real Robot

The simulation results suggest that our approach can lead to significant speed-ups in training as compared to learning only with raw images. But our simulation was not designed to be a perfect replication of the real-robot system. Firstly, the simulated parameters of the robot (e.g. actuator torques, joint frictions, etc.) are likely not closely matching the real robot. Secondly, sensors in simulation are prefect with no noise and missing observations, unlike in the real world. To gauge learning performance on the real robot compared to simulation, we first trained a a feature-based policy with feature only on the real robot.

We used a combination of tasks 1F, 2F, 3F, 4F, 5F and 8F together with the larger cup. Figure 4.13 shows the resulting average reward of task 5F. Note that only a single robot is used and training takes place with minimal human supervision (humans intervene only to reset the environment when the robot is unable to reset after multiple tries).

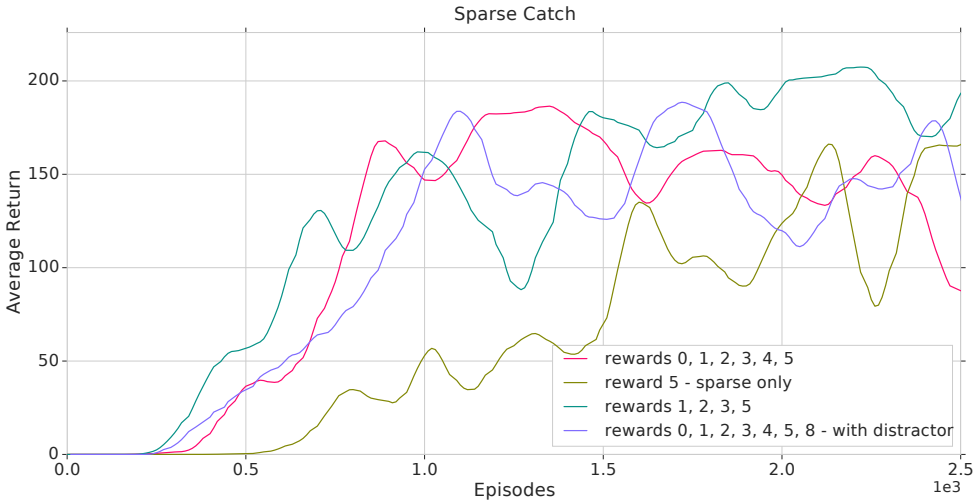


Figure 4.13: Real robot learning curve of task 5F for runs learning the Ball-in-a-Cup task with combinations of tasks 1F, 2F, 3F, 4F, 5F and 8F.

The yellow curve shows the performance of the policy for task 5F when only task 5F is used for training. This demonstrates learning is possible on the real robot with only a sparse reward, however, the other curves demonstrate that training speed can be improved with good auxiliary tasks. This is in agreement with the existing SAC-X work [80]. The purple curve shows the result of training with a purposefully useless "distractor" auxiliary task (8F). Despite this useless task, the robot learns a good sparse catch policy from features in approximately the same training time as when only useful tasks are provided to the agent. This suggests that while finding a minimal, optimal set of auxiliary tasks may be difficult, it may be safe to just add any additional auxiliary tasks one thinks may be useful, with minimal effect on training time and final policy performance.

Qualitatively, policies perform well. The swing-up is smooth and there is recovery from failed catches. Over 20 runs, each trial running for 10 seconds, we measured a 100% catch rate. The shortest time to catch being 2 seconds.

We repeated these experiments with the smaller cup, which increases the difficulty of the task,

in order to assess achievable precision and control. There was a slight slow-down in learning and a small drop in catch rate to 80%, still with a shortest time to catch of 2 seconds.

Learning with Different State-Spaces on a Real Robot

We applied our full method to obtain an image-based policy for the Ball-in-a-Cup task. Because of the long experiment times on a real robot, we opted to only use the approach that worked best in simulation: learning with different state-spaces combined with asymmetric actor-critic. We initially used tasks 1F, 2F, 3F, 4F, 5F, 1P, 2P, 3P, 4P, and 5P as in simulation. The task policies showed improvement, but learning was slower than in simulation. To minimize experimentation time we changed the auxiliary rewards into shaped rewards, with the intention of speeding-up training time for the auxiliary tasks so that good training data for the sparse catch task was more quickly generated.

Figure 4.14 shows the learning curve for task 5F and 5P when trained with 5F, 6F, 7F, 5P, 6P, and 7P. The critic filter vectors enabled only $S^{\sigma}_{\text{proprio}}$ and $S^{\sigma}_{\text{features}}$ for all tasks. Policies for both state-spaces converge to their final performance after around 5,000 episodes. This training time is longer than the equivalent simulation experiment. The increase in training time is likely due to differences in the quality of the simulated vs real-world camera images. The simulation images are noise-free, have no motion blur, and are consistently lit with proper colors. Also, the robot joint sensor have noise and delays which are not modeled in the simulation. Despite this, we are able to learn a sparse catch policy utilizing only image and proprioception data, from scratch in about 28 hours — ignoring time for episode resets. This is about twice as many episodes as in our successful simulation experiments.

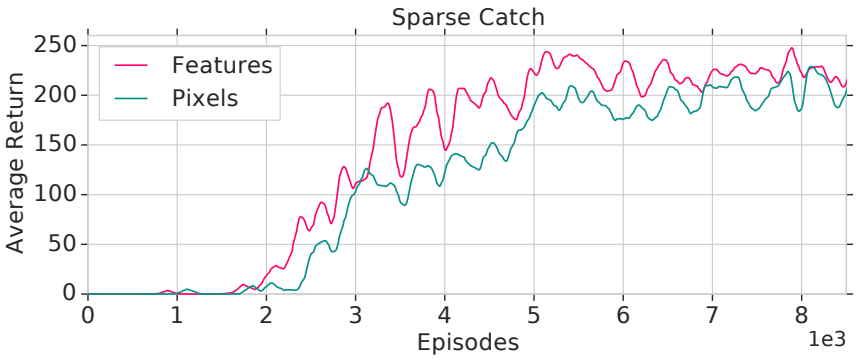


Figure 4.14: Learning curve for task 5F and 5P. Training was done with tasks 5F, 6F, 7F, 5P, 6P, and 7P. The asymmetric actor-critic technique was used.

We evaluated the policy learned after 8,360 episodes. During evaluation, exploration noise was set to zero. We ran a total of 300 episodes, achieving 100% catch rate (i.e. a catch occurred in all 300 episodes). Results of the evaluation, showing total reward and time to catch, are shown in Table 4.6.

Figure 4.15 shows a series of frames from a typical successful the execution of the pixel sparse reward policy. Figure 4.15a shows the robot running the reset procedure to begin a new episode. The robot dumps the cup to make sure the ball is hanging below before beginning.

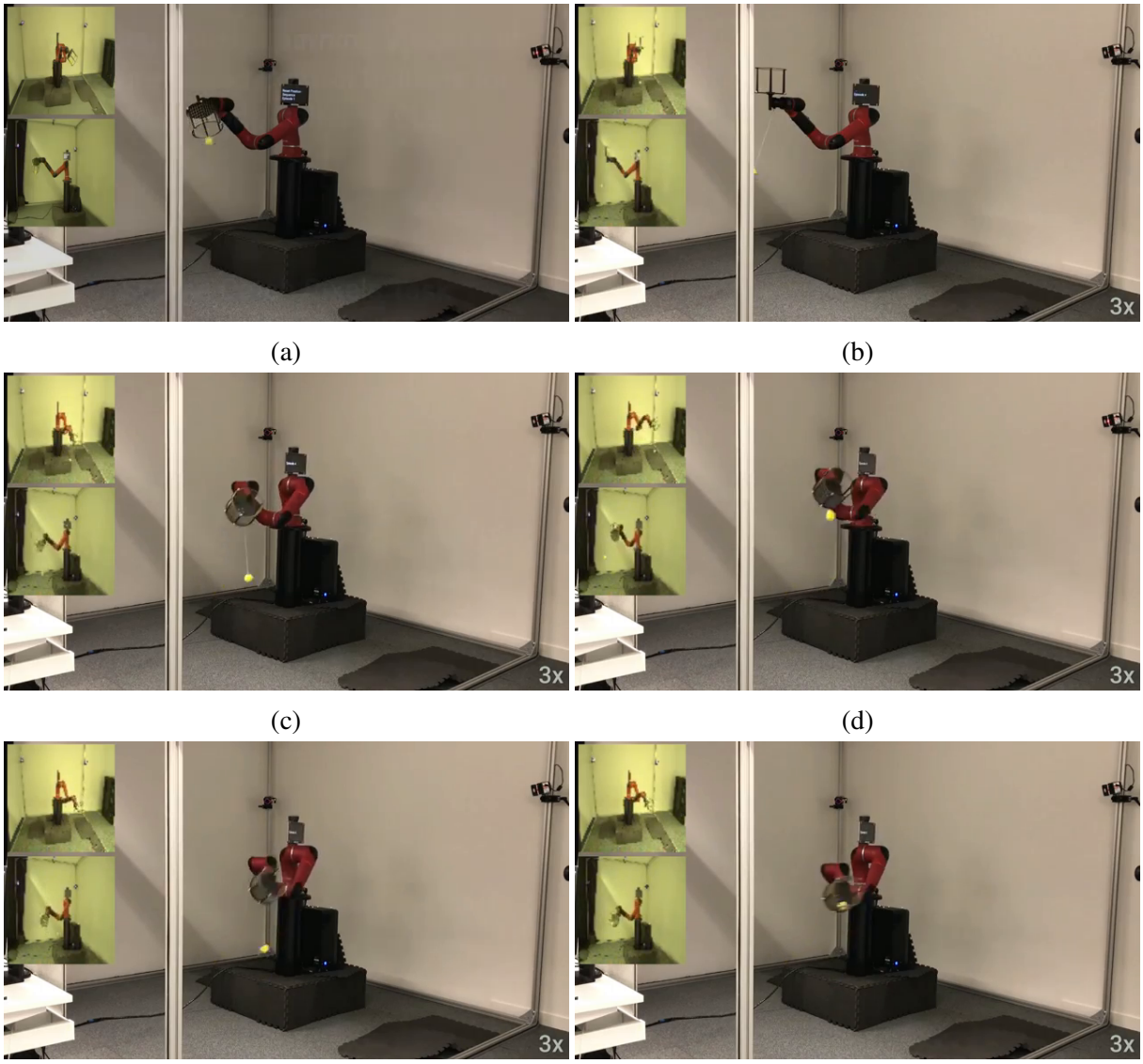
	Mean	Min.	Max.
Catch time (seconds)	2.39	1.85	11.80
Total reward (maximum of 500)	409.69	14.0	462.0

Table 4.6: Evaluation of pixel sparse catch policy over 300 episodes.

Figure 4.15b shows the start of the episode. Figure 4.15c shows the initial swing of the arm which begins building momentum in the ball. Figures 4.15d and 4.15e shows the arm making small swings with the correct timing to continue building the ball momentum. Figure 4.15f shows the frame just after the ball has entered the cup.

4.5 Summary

In this chapter we have introduced one of the major thesis contributions: how to learn simultaneously with multiple state representations. The goal of this technique to speed-up learning for high-dimensional state-spaces by providing distilled state representations at train time that are unavailable at test time. Such high-dimensional state spaces could be the tensor state-action spaces described in chapter 3 or raw sensors such as camera images. We show that our technique works with off-policy, probabilistic RL algorithms, allowing for further improvements as fundamental algorithm performance improves. We demonstrated that our technique works for simulated RoboCup SSL domains. Additionally, we showed that our technique works even on very different robots and tasks by learning the Ball-in-a-Cup task with a robot arm directly on real-hardware from scratch. This contribution of the thesis is not only generally applicable, but can be applied today on real-world robots.



(a) (b)

(c) (d)

(e) (f)

Figure 4.15: Sequence of frames taken from a run of the learned real Ball-in-a-Cup policy using images. Top left corner shows the two down-scaled images used by the agent.

Chapter 5

Related Work

In this chapter, we discuss related approaches to the two thesis contributed techniques: tensor state-action spaces as a transferrable representation and multi-task learning with different state spaces. The contributions in this thesis are compared to existing works with differences and improvements highlighted. In addition, we also discuss prior related work on Multi-Agent Reinforcement Learning (MARL) and RL work applied to the two major test domains used throughout this work: Ball-in-a-Cup and RoboCup SSL.

5.1 Transferrable Representations

This section provides an overview of transfer techniques and techniques related to our transferrable representations. A brief overview of related multi-agent learning and multi-agent transfer is also included, as we demonstrated our transferrable representation in chapter 3 on multi-agent domains.

5.1.1 Transfer Learning

Transfer learning is the study of how to take policies and knowledge from one environment and reuse the knowledge in a new, but similar environment in order to speed-up training or improve final policy performance [100]. There are many different approaches to transfer learning, with different assumptions about what properties of the environment can change including: state space, action space, reward function, transition dynamics, or a combination of different properties. This thesis focuses on designing a representation that can encode the assumptions about which aspects of the domain remain fixed, in order to make transferring learned policies easier.

Classic Transfer Learning Approaches

Just as there are many different assumptions about which parts of the environment can change, there are many different approaches to how to actually perform the transfer. One method is try and transfer partial policies between domains [5, 8, 54]. Typically, the agent, like in hierarchical RL, will try and learn options [93], with the change that these options will be reused in the transferred environment. Another related approach is to learn a similarity function for policies and

environments and then probabilistically transfer the policies to environments with high similarities [30]. Another option is to try and directly transfer information about environments or regions of environments with similar dynamics [57]. If these regions can be identified then the collected exploration data can be reused in the new environment. Another approach is to use experience in a new domain to correct a model learned in a previous domain [119]. This corrective data can come through new data collection or a human teaching signal.

A method related to this thesis approach of finding transferrable representations is the use of proto-value functions (PVF) for transfer [28, 63]. Proto-value functions is a method of learning a set of orthogonal basis functions tailored for an environment that can then be used with a linear weighted sum policy. In the context of transfer, a PVF can be learned in one environment and then reused for training in a new environment [28]. By starting with a good representation in the new environment the learning can be improved. While both this thesis and this work look at transfer learning using good representations, the approaches are significantly different. PVF are designed for linear weighted sum policies, whereas this work looks at representations in a deep learning context. This work also explores transfer in multi-agent contexts and with different assumptions about which parts of the environment are changing.

Simulation to Real (Sim2Real) Transfer

In robotics, a lot of effort has been put into exploring training in simulation and then executing policies on real robot hardware. This approach is commonly known as Sim2Real [84]. There are many approaches that attempt to completely eliminate the differences between simulation and the real environment. If this is perfectly achieved, then nothing related to transfer learning is being done. However, there are two commonly used techniques which do bring in an element of transfer learning: domain randomization during training [72, 106] and additional training after simulation in the real-world [84]. Domain randomization attempts to account for the differences between the training simulation environment and the transferred to real-world by training with many different perturbations to the physics settings such as coefficients of frictions, actuation forces, and even integration time-steps [72]. Given that the agent is training across many similar related tasks in parallel, this can also be viewed as multi-task learning [27, 46, 73].

Meta-Learning

A more modern technique, meta-learning, is also related to transfer learning. Unlike transfer learning, which attempts to leverage previously used information from an environment in a new environment, meta-learning attempts to learn how to improve the learning algorithm itself [6, 24, 31, 48]. One approach uses a neural network to do the gradient descent optimization [6, 48]. This network can be optimized to perform updates which lead to faster convergence. Another approach is the Model-Agnostic Meta-Learning (MAML) class of algorithms, which attempt to optimize a network so that when the network is applied on subsequent problems the number of optimization updates needed is minimized [31].

Relational Reinforcement Learning (RRL)

Relational Reinforcement Learning (RRL) is a different way of defining an MDP and RL framework where the transitions and rewards are defined in an agent and object centered way [23, 25, 38]. Environments consist of a set of classes, each with multiple instances with their own state variables. Transitions and rewards are defined as functions on class instances. The goal is to learn value functions for the different relations which can then be composed to find a policy for the specific instance of the environment. Previous work has explored this concept as a means to transfer policies across varying amounts of agents and objects in an environment [38]. However, this work assumed that value functions were composable and that the transitions and rewards could be defined in a relational way. This thesis does focus on transferrable representations that can accomplish similar goals (e.g. transfer a policy to different numbers of agents and objects), but it does not require a relational MDP formulation.

5.1.2 Multi-Agent Reinforcement Learning (MARL)

Multi-Agent Reinforcement Learning (MARL) is reinforcement learning when there are multiple agents learning policies in the same environment [18]. Each agent needs to learn not only how to accomplish it's goals but how to adapt to the policies of the other agents in the environment. These other agents may be adversarial or cooperative or a mix of both. Recently there have been a few high profile successes in multi-agent learning including human-level performance in first-person capture the flag [50], the Dota 2 OpenAI Five Challenge [71], and the StarCraft II AlphaStar bot [112]. While this thesis does not focus on multi-agent learning specifically, some of the transferrable representation work presented in chapter 3 is designed to work in multi-agent environments. Therefore, this section contains a high-level overview of MARL and multi-agent transfer learning techniques.

There are many existing works on multi-agent learning. The most straightforward approach in a cooperative environment is to find a centralized policy for the MDP [18]. This policy will take in a Cartesian product of each agent state and output a Cartesian product of actions for each agent. While this approach works, it can be quite expensive in terms of computation and training data as the dimensionality of the state and action space grows quickly for each new agent. It also is not suited for environments where the internal states of the other agents are unknown or uncontrollable.

Another method of multi-agent learning is to naïvely apply an RL independently to each agent in the environment pretending that the other agents are static and part of the environment [18]. While this approach can work in some cases, the non-stationarity of the other agents breaks the assumptions of standard RL algorithms and can lead to a failure to learn. One method to rectify this is to do importance sampling on the training data [32]. Data where the recorded transition actions of the other agents match the current agent policies are unmodified, while actions that are drastically different to the current policy are down-weighted. This has successfully been applied to small StarCraft inspired domains [32].

A different approach is to allow agents to communicate with one another, either through a learned language or a fixed vocabulary [33]. Before choosing an action, each agent can communicate with its peers. By sharing information with their peers the agents can learn about the

internal states of the other agents and adjust their behavior accordingly. It is also possible to share some of the internal hidden layers to help coordinate agents [97].

An alternative to sharing at test time is to share at training time. In an actor-critic setup, it is possible to use the asymmetric actor-critic technique [74] to provide the critic with the states and actions of the other agents [34, 62]. This allows the critic to learn a good estimate of the value of an individual agents actions for a state, which leads to better optimization of the policy.

More game theoretic approaches are also possible, especially in adversarial environments such as StarCraft II. Bowling and Veloso [12] presented a method of adapting learning rates in a multi-agent context such that under certain conditions a Nash equilibrium strategy would be found by the agents. AlphaStar used a tournament self-play training to slowly ramp-up the complexity and quality of the trained agent strategies. New opponents in each tournament round were created by branching off of successful versions of the agent from the previous tournament rounds.

Some approaches such as the OpenAI Five [71] and First-person capture the flag [50] make use of reward schemes where individual agents are incentivized to cooperate by a team reward, while learning individual behaviors with an agent specific intrinsic reward. This approach is often combined with one of the above approaches in order to have stable learning.

5.1.3 Multi-agent Transfer Learning

In this thesis we have applied our transferrable representations to MARL and then transfer these policies as the number of agents in the environment vary. Work in Relational Reinforcement Learning (RRL) is most similar to this approach [23, 25, 38] (see section 5.1.1). Because of the way transitions and rewards and value functions defined based on classes of objects, it is possible to vary the number of instances of the classes without having to relearn the value functions and policies. However, as mentioned, this approach requires the MDP be defined as a relational MDP which differs from standard formulations. It also placed requirements on the composability of the sub-value functions.

Overall, the subject of multi-agent transfer learning has not been the subject of much investigation. Most of the techniques are adaptations of single agent transfer learning to multi-agent scenarios. Adapted techniques include: object oriented MDPs [22], task mappings [11], experience sharing [99], supervision from more experienced agents [36].

5.2 Simultaneous Representations

In this section we explore techniques related to our approach of simultaneously learning policies with different state spaces. We discuss auxiliary reward task learning, curriculum learning, multi-task learning, and behavior cloning as related techniques.

5.2.1 Auxiliary reward tasks

The approach of learning multiple policies with different state and action spaces is an extension of work using auxiliary tasks for training. Typically, previous work has restricted the auxiliary

tasks to different reward functions or loss functions [49]. One of the earliest examples of this approach is the idea of General Value functions Sensorimotor streams [98] This idea was extended to Deep RL in work on Universal Value Function Approximators [85], which learns a goal conditioned action-value function. Each goal can be considered an auxiliary task. Hindsight Experience Replay (HER) [7] introduced the idea of computing rewards for different goals "in hindsight" by treating the end of the recorded trajectory as the intended goal. Scheduled Auxiliary Controller (SAC-X) [80], which this thesis extends, generalized this idea by learning multiple "intention policies" for different semantically grounded, auxiliary tasks. Each of the intention policies is executed to drive exploration for the other intention policies.

5.2.2 Curriculum learning

Our approach can also be related to curriculum learning approaches [9]. Curriculum learning tries to speed-up learning and improve the final learned policy by starting the training on simple tasks, and slowly increasing the difficulty/complexity over time. This technique was an important component of letting a variety of simulated agents learn to walk over difficult terrain [45]. The training course used by the walking agents started smooth, with easy obstacles, and as the agent moved farther from the start, the obstacles got more and more difficult to navigate.

Curriculum learning approaches typically track internal metrics such as improvement in reward, or an intrinsic motivation reward signal. These metrics are used to either schedule difficulty changes or invent new training tasks [35, 52, 86]. Unlike some other curriculum approaches, we restrict our available tasks to a pre-defined set. While SAC-X can do adaptive scheduling of the different tasks, in this thesis we focus on the simple case of uniform random task selection.

5.2.3 Multi-task learning

Our approach is also related to multi-task learning, which has been studied in numerous other contexts [27, 46, 73]. However, prior approaches have mostly focused on varying reward functions and varying environment behaviors. Whereas the state space and action spaces were the same. For example, multi-task learning across multiple Atari games [27], while the content in the images used as the state are different for each game, the states are still images containing Atari graphics. Additionally, the action space of the controller does not change between games. Work on multi-task learning with varying observations is more scarce. The most similar approach to the one presented in this thesis is the asymmetric actor-critic technique [72, 74]. In this technique the critic network receives additional state-input which is only available at training time, while the actor receives only the state observations used in the final test time policy. The goal is to ease credit assignment by providing additional context in the critic to help when training with policies that use images as the state. The work in this thesis generalizes this idea to different states not only between the actor and critic, but also across different tasks. It also explores different action spaces between different tasks.

5.2.4 Behavior cloning

Our approach has weak connections to behavior cloning [1, 58, 59, 82], a form of imitation learning where an agent attempts to replicate another agent's policy. In this thesis, we show how a policy can be learned that works with multiple state and action spaces. Typically, one state space will be a feature-based representation for the task, and the other state space will be a raw sensor based representation, such as images. Once the agent has learned a reasonable behavior in the feature-based state space, this can be seen as generating demonstrations for the vision based policies (or vice versa depending on which policy improves first). However, unlike techniques like Guided Policy Search (GPS) [58, 59], which then treat the policy learning as a supervised learning problem which attempts to duplicate the same behavior, we allow the different state spaces to learn completely different behaviors. Rather than maximizing the similarity between policies in each state-action space, we are still interested in learning the optimal policy according to the transition samples collected.

5.3 Environments

Reinforcement Learning has been applied successfully to many environments. From classic board games like Go [91, 92] to video games like Atari [66, 67] and even to simulated and real-world robotic tasks like flying a model helicopter [2], robot soccer [78, 95], flipping pancakes [55], manipulating objects [3, 72], walking [45], and Ball-in-a-cup with a robot arm [19, 53]. Given the wide variety in domains, there are many domains this thesis could demonstrate its approach with. However, for the most part we choose to focus on dynamic tasks.

Dynamic robotic tasks have additional difficulties when compared to static robotic tasks. Goal states are often located in unstable parts of the state space that are difficult to reach (and stay in) with random exploration. Additionally, the minimum control rates necessary to achieve good performance usually are higher due to these instabilities. Nonetheless, reinforcement learning for dynamic robot tasks has been performed in the past on tasks such as pancake flipping [55], walking on unstable surfaces [40], and as used in this thesis the Ball-in-a-Cup task [19, 53].

5.3.1 Ball-in-a-Cup

As described in previous chapters and sections, the Ball-in-a-Cup game consists of a robot arm holding a cup with a ball hanging below attached via a string. The goal being to swing the ball up and catch it in the cup. This domain has been previously studied in the context of robotic reinforcement learning [53]. Chiappa et al. [19] complete the task with a simulated robot arm using motion templates generated from human demonstration data. More directly related, Kober and Peters [53] complete the Ball-in-a-Cup task on a real robot but their required imitation learning through kinestetic teaching, reward shaping, expressive features, and a policy class restricted to Dynamic Movement Primitives (DMP). However, neither of these techniques work directly with images and learn directly on real robot hardware from scratch like the techniques presented in chapter 4.

5.3.2 Robot Soccer Domains

As discussed in chapter 2, RoboCup soccer, and specifically RoboCup SSL is a well studied task in robotics and RL research. Due to scalability of existing techniques, there has not been any large successful work attempting to learn RoboCup soccer end-to-end from scratch. Instead, different pieces of robot soccer game play can be broken into smaller test tasks that highlight different aspects of the game from simple navigation control tasks to multi-agent ball manipulation in games such as Keepaway [95, 96]. These types of environments have also been proposed as benchmark tasks for reinforcement learning [94, 96]. This has lead to a number of works applying reinforcement learning techniques to RoboCup soccer domains [30, 78, 79, 95, 107].

Outside of RL, RoboCup SSL, and more specifically the CMDragons' robots have been used in a large variety of research. [17, 64] This includes research into computer vision to design the original butterfly pattern used by the SSL vision software [15]. There has been research into motion planning for the robots which this thesis used in baselines and for things like episode resets. [14, 16]. Research into physics based modeling of the robots helped to create the physics simulations used in the simulation environments. [117, 118]. Non-RL learning has been performed with the robots including bandit optimizations for free-kick tactic selection and learning to adapt to opponent behaviors. [26, 65]. The CMDragons' robots have also been used as a platform for planning research in adversarial environments. [10, 21]

The following sections delve deeper into two domains related to the domains used in this thesis.

Keepaway

Keepaway is a subdomain inspired by RoboCup robot soccer. It is multi-agent and can be partially observable (with each agent having a limited sensor view of the environment) [95, 96]. The game consists of two teams of agents. One team starts with the ball, and attempts to maintain control of the ball as long as possible by dribbling and passing to teammates. An opponent team of agents attempts to intercept and take the ball from the opposing team as quickly as possible.

Stone et al. [95] utilized an options [93] like framework with hand-coded low-level controls. Agents learned to call these sub-policies at appropriate times in scenarios with 4 teammates vs 3 opponents. A significant effort went into determining the proper low level actions and state-features to utilize.

This domain has also been used in the context of transfer learning. This was one of the test domains used by Fernández et al. [30] in their policy library transfer work.

Ball Manipulation Skills

In addition to multi-agent domains like Keepaway other aspects of robot soccer have been studied, such as ball-manipulation skills. These may include dribbling (moving the ball in a controlled manner through physical contact), passing the ball between robots, and shooting goals. Many of these skills have been studied before in an RL context. Riedmiller et al. [79] used RL on real robots for the Middle Size League in order to learn ball manipulation skills. Recent Deep RL work from Hausknecht et al. [43] was able to learn, in simulation, to navigate to a ball and

score on an empty goal using imitation learning. This was later refined to learn, in simulation, from scratch [42].

Chapter 6

Conclusion and Future Work

In this chapter, we review the contributions of this thesis. We then discuss future avenues of research that can build on the foundations laid by this thesis. Finally, we summarize the entire thesis document.

6.1 Contributions

As laid out in chapter 2, two of the major issues still holding back robot Deep RL are: sample complexity and rigidity of learned policies. In this thesis we have shown two major techniques that improve the state of the art in both of these categories: tensor state-action spaces and auxiliary task learning with multiple state representations.

Transferrable Tensor State-Action Spaces We introduced the concept of designing a transferrable state-action space using multi-dimensional tensors that allows for zero-shot transfer as the number of agents and objects in an environment vary. By anticipating the types of changes to the environment, we showed how a standard feature representation and parameterized action space can be converted back and forth between a tensor representation where the amount of parameters remain fixed regardless of how many agents or other objects are added and removed from the field. We presented a specific mapping for environments where robot positioning is key to good performance. Using this representation we showed how to train and apply both single and Multi-Agent Reinforcement Learning (MARL) across a number of different simulated environments inspired by RoboCup, a modified version of Atari breakout, and execution on real SSL robots.

Fully Convolutional Q-Network (FCQN) We introduced a new policy network architecture called the Fully Convolutional Q-Network (FCQN), which uses only convolution and deconvolution operations, rather than the more common fully connected layers. This network architecture is designed to work with the tensor state-action spaces, taking advantage of assumptions about locality and translational invariance built into convolution and deconvolution operations. More importantly, this representation allows for zero-shot transfer of both single and multi-agent

policies across environment sizes. We showed that this architecture works for both simulated RoboCup inspired domains, a modified version of Atari breakout, and real SSL robot execution.

Auxiliary Task Learning with Multiple State Representations We introduced an extension of the Scheduled Auxiliary Controller (SAC-X) framework that allows for learning simultaneously with multiple state representations. Doing so allows for using different features at train and test time. Additionally, at train time lower-dimensional, task specific representations can be used, which can speed up learning compared to learning with just the high-dimensional image state space. We showed that this technique works on both simulated RoboCup and real-world Ball-in-a-Cup with a robot arm. Using this technique we were able to quickly learn a policy using multiple color images, directly on a real-robot.

Experimental Results on Multiple robot tasks All of the work presented in this thesis was designed with real robot applications in mind. We showed in chapter 2 that while existing Deep RL can be applied to robots, it has a number of short-comings. We demonstrated that our transferrable representation and FCQN architecture policies work on real robot hardware despite the high-level, abstractness of the representation. We demonstrated our auxiliary task framework technique with both simulated and real robots, including the real-world Ball-in-a-Cup with a robot arm task.

6.2 How Contributions Improve on Shortcomings

As discussed in Chapter 2, existing approaches to robot deep reinforcement learning have a number of shortcomings that we have worked to address in this thesis. We now summarize how our contributions attack these shortcomings.

Sample Complexity and Training Time The thesis technique of auxiliary task learning with multiple state representations significantly reduces the required number of samples and training time required to learn sparse-reward policies in high-dimensional state spaces. As shown, this technique made a previously unsolvable task, learning Ball-in-a-Cup with images directly on a robot from scratch, into something solvable with about 23 hours of active data collection time. Our simulation results showed how much of an improvement the contributed technique can give.

The contributed tensor state-action space with the FCQN network also contributed to reducing sample complexity and training time. In our experiments, we showed that compared to the tensor state-action space without the FCQN and a traditional state-action space with a standard RL policy network, our approach converged significantly faster.

The tensor state-action space also contributes to a reduction of sample complexity and training time, by allowing for zero-shot transfer of policies. As discussed, this means the initial training cost is amortized across a wide variety of changed environments. Rather than having to train new policies for each change, a single policy can be used.

Adaptability The tensor state-action space with the FCQN network contributes the most to improving adaptability of learned policies. The technique was specifically designed to make zero-shot transfer across changes to environment size, number of objects, and number of agents possible. A standard approach would have required training from scratch whenever one of these properties changed.

The auxiliary task learning with multiple state representations also has some contributions towards adaptability. The actor and critic network architecture allows the agent to learn policies that can rely on different subsets of the available observations, which allows the tasks to operate with different state spaces. This adaptability to different state spaces makes it easier to introduce new sensor types when applying RL to robots.

Safety Finally, while the thesis did not focus specifically on safety, we did introduce throughout the main thesis and appendices practical safety algorithms for our various robots. These included algorithms to keep the soccer robots confined to a field area, physical modifications to the environment, and a low-pass filter on the Ball-in-a-Cup robot arm to protect it from exploratory, high-frequency actions.

Additionally, our techniques do improve on scalability and adaptability. By reducing training time and samples, agents converge to reasonable, safe behavior much faster, reducing the time for mistakes. By adapting zero-shot across changes to the environment, reasonable behavior is maintained despite the environment changes.

6.3 Future Work

In this section, we discuss some possible future research directions that can build off the contributions of this thesis.

Alternative Transferrable Representations We showed how image-like, multi-dimensional tensors can be used to do zero-shot transfer as the number of agents and objects vary and as the environment size varies. However, for extremely sparse environments, this representation can be inefficient. Other representations such as set based representations like those used in point-cloud processing should be investigated. [76] Like the tensor state-action spaces, work would need to be done to determine how to map between this new representation and the existing state-action space for efficient learning.

Alternative Transferrable Policy Network Architectures We showed how the FCQN architecture works well with our tensor state-action space representation and facilitates zero-shot transfer across environment sizes. If other transferrable representations are developed, it makes sense that there are likely new transferrable policy network architectures that are associated with these new representations. Network architectures for set based states, such as point clouds, can be investigated. [76]. Sequence processing architectures such as RNNs and ideas from Natural Language Processing (NLP) such as transformer architectures can be investigated. [20, 47, 108] The idea is to find representations that can deal with arbitrarily sized inputs and outputs without requiring new parameters be learned.

Auxiliary Task Learning with Multiple Action Representations We showed that there are major benefits to simultaneously learning auxiliary tasks with multiple state spaces. It seems likely that we can gain additional benefits for expanding the auxiliary task framework to accommodate tasks with different action spaces. This could allow for learning across multiple robot morphologies simultaneously. Assuming some similarities, such as a common mobile base with different arms, the action spaces would share some common components and training data from multiple robots may lead to speed-ups or improvements in robustness of the policies. Even with identical robots, different control spaces can be used (e.g. torque control vs velocity control) and using sensors and models of the robot we can collect data in one action space and use it to generate data in a different action space. Finally, we could potentially have different action spaces for the same robot that account for different parts breaking. For example, the soccer robots have four omni-wheels but technically can drive with only three, being able to simultaneously train policies for the ideal case of all wheels working and the backup case of specific wheels failing could be useful.

Auxiliary Task Learning with Multiple Time-Scales We could potentially take advantage of learning with different time-scales across auxiliary tasks. Many times, due to the difficulty of credit assignment in long sequences, we will run the agents at a slower control frequency than the physical hardware and sensors are capable of. For mostly static tasks, this is generally not an issue, but for dynamic tasks and environments, the faster the robot can execute the better it can control and react to changes. We could potentially have some tasks run at a slow, easier to learn but not optimal, time-scale, while still collecting data at the maximum sensing rate. This additional data could be used as good training data to speed-up training on the maximum time-scale policies.

6.4 Summary

In this thesis we introduced two main techniques for robot deep reinforcement learning: transferrable tensor state-action spaces and auxiliary task learning with multiple state representations. These two contributions help reduce training time and sample complexity and improve policy flexibility for real robots. The tensor state-action space representation allows one to convert an existing task environment to a representation that can transfer zero-shot across changes to the number of agents and objects in the environment. This can help amortize any expensive training cost across many different similar environments. We showed how this representation can combine with a new policy network architecture, the Fully Convolutional Q-Network (FCQN), to provide zero-shot transfer across environment sizes. This can allow for not only flexibility, but the ability to train in smaller, less resource intensive environments and then transfer to full-scale without additional training. We showed how one can learn multiple auxiliary tasks simultaneously that differ not only in reward function but in the state space. This allows for training policies that utilize different inputs at test and train time. Additionally, we showed that this can significantly speed-up training for high-dimensional state spaces, such as raw image sensors. We demonstrated all of these techniques across a variety of simulated and real-robot tasks including RoboCup SSL tasks and the Ball-in-a-Cup game with a robot arm.

Bibliography

[1] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1. ACM, 2004. 5.2.4

[2] Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Y Ng. An application of reinforcement learning to aerobatic helicopter flight. In *Advances in neural information processing systems*, pages 1–8, 2007. 1.1, 5.3

[3] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019. 2.1.3, 5.3

[4] Eitan Altman. *Constrained Markov decision processes*, volume 7. CRC Press, 1999. 2.1.3

[5] David Andre and Stuart J Russell. State abstraction for programmable reinforcement learning agents. In *AAAI/IAAI*, pages 119–125, 2002. 5.1.1

[6] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016. 5.1.1

[7] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058, 2017. 5.2.1

[8] Mehran Asadi and Manfred Huber. Effective control knowledge transfer through learning skill and representation hierarchies. In *IJCAI*, volume 7, pages 2054–2059, 2007. 5.1.1

[9] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009. 5.2.2

[10] Joydeep Biswas, Juan Pablo Mendoza, Danny Zhu, Benjamin Choi, Steven Klee, and Manuela Veloso. Opponent-Driven Planning and Execution for Pass, Attack, and Defense in a Multi-Robot Soccer Team. In *Proceedings of AAMAS'14, the Thirteenth International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Paris, France, May 2014. 5.3.2

[11] Georgios Boutsioukis, Ioannis Partalas, and Ioannis Vlahavas. Transfer learning in multi-

agent reinforcement learning domains. In *European Workshop on Reinforcement Learning*, pages 249–260. Springer, 2011. 5.1.3

[12] Michael Bowling and Manuela Veloso. Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136(2):215–250, 2002. 5.1.2

[13] Brett Browning, James Bruce, Michael Bowling, and Manuela Veloso. STP: Skills, tactics and plays for multi-robot control in adversarial environments. *Journal of Systems and Control Engineering*, 219:33–52, 2005. The 2005 Professional Engineering Publishing Award. 2.2.3

[14] James Bruce and Manuela Veloso. Real-time randomized path planning for robot navigation. In *Proceedings of IROS-2002*, Switzerland, October 2002. 2.2.3, 5.3.2

[15] James Bruce and Manuela Veloso. Fast and accurate vision-based pattern detection and identification. In *Proceedings of ICRA'03, the IEEE International Conference on Robotics and Automation*, Taiwan, May 2003. 2.2.1, 2.2.1, 5.3.2

[16] James Bruce and Manuela Veloso. Real-time randomized motion planning for multiple domains. In *Proceedings of the RoboCup Symposium 2006*, Bremen, Germany, June 2006. 2.2.3, 5.3.2

[17] James Bruce, Stefan Zickler, Michael Licitra, and Manuela Veloso. CMDragons: Dynamic Passing and Strategy on a Champion Robot Soccer Team. In *Proceedings of ICRA'2008*, Pasadena, CA, 2008. 1.4, 2.2.1, 5.3.2

[18] Lucian Bu, Robert Babu, Bart De Schutter, et al. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(2):156–172, 2008. 5.1.2

[19] Silvia Chiappa, Jens Kober, and Jan R Peters. Using bayesian dynamical systems for motion template libraries. In *Advances in Neural Information Processing Systems*, pages 297–304, 2009. 5.3, 5.3.1

[20] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014. 6.3

[21] Philip Cooksey, Juan Pablo Mendoza, and Manuela Veloso. Opponent-Aware Ball-Manipulation Skills for an Autonomous Soccer Robot. In *Proceedings of the RoboCup Symposium*, Leipzig, Germany, July 2016. Springer. Nominated for Best Paper Award. 5.3.2

[22] Felipe Leno Da Silva and Anna Helena Reali Costa. Transfer learning for multiagent reinforcement learning systems. In *IJCAI*, pages 3982–3983, 2016. 5.1.3

[23] Carlos Diuk, Andre Cohen, and Michael L Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pages 240–247. ACM, 2008. 5.1.1, 5.1.3

[24] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016. 5.1.1

[25] Sašo Džeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine learning*, 43(1-2):7–52, 2001. 5.1.1, 5.1.3

[26] Can Erdogan and Manuela Veloso. Action Selection via Learning Behavior Patterns in Multi-Robot Systems. In *Proceedings of IJCAI'11, the 20th International Joint Conference on Artificial Intelligence*, Barcelona, Spain, July 2011. 5.3.2

[27] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018. 4.2.3, 5.1.1, 5.2.3

[28] Kimberly Ferguson and Sridhar Mahadevan. Proto-transfer learning in markov decision processes using spectral methods. *Computer Science Department Faculty Publication Series*, page 151, 2006. 5.1.1

[29] Fernando Fernandez and Manuela Veloso. Learning Domain Structure through Probabilistic Policy Reuse in Reinforcement Learning. *Progress in Artificial Intelligence*, 2012. DOI:10.1007/s13748-012-0026-6. 2.3.1

[30] Fernando Fernández, Javier García, and Manuela Veloso. Probabilistic policy reuse for inter-task transfer learning. *Robotics and Autonomous Systems*, 58(7):866–871, 2010. 2.2.2, 5.1.1, 5.3.2

[31] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR. org, 2017. 5.1.1

[32] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip HS Torr, Pushmeet Kohli, and Shimon Whiteson. Stabilising experience replay for deep multi-agent reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1146–1155. JMLR. org, 2017. 5.1.2

[33] Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. 2016. URL `http://arxiv.org/abs/1605.06676v2`. 5.1.2

[34] Jakob N Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018. 5.1.2

[35] Sébastien Forestier, Yoan Mollard, and Pierre-Yves Oudeyer. Intrinsically motivated goal exploration processes with automatic curriculum learning. *arXiv preprint arXiv:1708.02190*, 2017. 5.2.2

[36] Daniel Garant, Bruno Castro da Silva, Victor Lesser, and Chongjie Zhang. Accelerating multi-agent reinforcement learning with dynamic co-learning. Technical report, Technical report, 2015. 5.1.3

[37] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010. 3.4.1

[38] Carlos Guestrin, Daphne Koller, Chris Gearhart, and Neal Kanodia. Generalizing plans to new environments in relational mdps. In *Proceedings of the 18th international joint conference on Artificial intelligence*, pages 1003–1010. Morgan Kaufmann Publishers Inc., 2003. 5.1.1, 5.1.3

[39] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018. 3.4, 4.2.1

[40] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018. 3.4, 4.2.1, 5.3

[41] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, 2015. URL `http://arxiv.org/abs/1509.06461v3`. 3.4, 3.4.1, 3.6.2, A.3

[42] Matthew Hausknecht and Peter Stone. Deep reinforcement learning in parameterized action space. In *Proceedings of the International Conference on Learning Representations (ICLR)*, May 2016. 2.2.2, 3.2.1, 5.3.2

[43] Matthew Hausknecht, Yilun Chen, and Peter Stone. Deep imitation learning for parameterized action spaces. In *AAMAS Adaptive Learning Agents (ALA) Workshop*, May 2016. 2.2.2, 5.3.2

[44] Nicolas Heess, Gregory Wayne, David Silver, Timothy Lillicrap, Tom Erez, and Yuval Tassa. Learning continuous control policies by stochastic value gradients. In *Advances in Neural Information Processing Systems*, pages 2944–2952, 2015. 4.2.1

[45] Nicolas Heess, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, Ali Eslami, Martin Riedmiller, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017. 5.2.2, 5.3

[46] Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech Czarnecki, Simon Schmitt, and Hado van Hasselt. Multi-task deep reinforcement learning with popart. *arXiv preprint arXiv:1809.04474*, 2018. 5.1.1, 5.2.3

[47] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 6.3

[48] Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001. 5.1.1

[49] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016. 5.2.1

[50] Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in first-person multiplayer games with population-based deep reinforcement learning. *arXiv preprint arXiv:1807.01281*, 2018. 5.1.2

[51] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016. 3.4

[52] Lu Jiang, Zhengyuan Zhou, Thomas Leung, Li-Jia Li, and Li Fei-Fei. Mentornet: Learning data-driven curriculum for very deep neural networks on corrupted labels. In *International Conference on Machine Learning*, pages 2309–2318, 2018. 5.2.2

[53] Jens Kober and Jan Peters. Learning motor primitives for robotics. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 2112–2118. IEEE, 2009. 5.3, 5.3.1

[54] George Konidaris and Andrew G Barto. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, volume 7, pages 895–900, 2007. 5.1.1

[55] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. Robot motor skill coordination with em-based reinforcement learning. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 3232–3237. IEEE, 2010. 5.3

[56] Steven M LaValle and James J Kuffner Jr. Randomized kinodynamic planning. *The International Journal of Robotics Research*, 20(5):378–400, 2001. 2.2.3

[57] Alessandro Lazaric. *Knowledge transfer in reinforcement learning*. PhD thesis, PhD thesis, Politecnico di Milano, 2008. 5.1.1

[58] Sergey Levine and Pieter Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In *Advances in Neural Information Processing Systems*, pages 1071–1079, 2014. 5.2.4

[59] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016. 5.2.4

[60] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *Internal Conference on Learning Representations*, 2016. URL `http://arxiv.org/abs/1509.02971v5`. 2.2.4, A.2

[61] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015. 3.4.3

[62] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*, pages 6382–6393, 2017. 5.1.2

[63] Sridhar Mahadevan and Mauro Maggioni. Proto-value functions: A laplacian framework for learning representation and control in markov decision processes. *Journal of Machine Learning Research*, 8(Oct):2169–2231, 2007. 5.1.1

[64] Juan Pablo Mendoza, Joydeep Biswas, Danny Zhu, Philip Cooksey, Richard Wang, Steven Klee, and Manuela Veloso. Selectively Reactive Coordination for a Team of Robot Soccer Champions. In *Proceedings of AAAI'16, the Thirtieth AAAI Conference on Artificial Intelligence*, Phoenix, AZ, February 2016. 1.4, 2.2.1, 5.3.2

[65] Juan Pablo Mendoza, Manuela Veloso, and Reid Simmons. Online Learning of Robot Soccer Free Kick Plans using a Bandit Approach. In *Proceedings of ICAPS'16, the 26th International Joint Conference on Automated Planning and Scheduling*, London, UK, June 2016. 5.3.2

[66] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013. URL `http://arxiv.org/abs/1312.5602v1`. 5.3

[67] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015. 3.6.1, 3.6.3, 5.3

[68] Anahita Mohseni-Kabir, Manuela Veloso, and Maxim Likhachev. Efficient robot planning for achieving multiple independent partially observable tasks that evolve over time. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 202–211, 2020. 2.3.1

[69] Rémi Munos, Tom Stepleton, Anna Harutyunyan, and Marc Bellemare. Safe and efficient off-policy reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1054–1062, 2016. 4.2.1, 4.2.3, 4.2.3

[70] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999. 4.2.6

[71] OpenAI. Openai five. `https://blog.openai.com/openai-five/`, 2018. 5.1.2

[72] OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *arXiv preprint arXiv:1808.00177*, 2018. 2.1.3, 5.1.1, 5.2.3, 5.3

[73] Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015. 5.1.1, 5.2.3

[74] Lerrel Pinto, Marcin Andrychowicz, Peter Welinder, Wojciech Zaremba, and Pieter Abbeel. Asymmetric actor critic for image-based robot learning. *arXiv preprint arXiv:1710.06542*, 2017. 4.2, 4.2.5, 5.1.2, 5.2.3

[75] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014. 2.2.4

[76] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017. 6.3

[77] Alex Ray, Joshua Achiam, and Dario Amodei. Benchmarking safe exploration in deep reinforcement learning. 2.1.3

[78] Martin Riedmiller and Thomas Gabel. On experiences in a complex and competitive gaming domain: Reinforcement learning meets robocup. In *2007 IEEE Symposium on*

Computational Intelligence and Games, pages 17–23. IEEE, 2007. 2.2.2, 5.3, 5.3.2

[79] Martin Riedmiller, Thomas Gabel, Roland Hafner, and Sascha Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, 2009. 1.1, 2.2.2, 5.3.2

[80] Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Van de Wiele, Volodymyr Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing-solving sparse reward tasks from scratch. *arXiv preprint arXiv:1802.10567*, 2018. 2.3.2, 4.1, 4.2.1, 4.2.3, 4.2.6, 4.4.2, 5.2.1

[81] Raul Rojas and Alexander Gloye Förster. Holonomic control of a robot with an omnidirectional drive. *KI-Künstliche Intelligenz*, 20(2):12–17, 2006. 2.2.1

[82] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. 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*, pages 627–635, 2011. 5.2.4

[83] Andrei A Rusu, Matej Vecerik, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. Sim-to-real robot learning from pixels with progressive nets. *arXiv preprint arXiv:1610.04286*, 2016. 2.1.3

[84] Andrei A. Rusu, Matej Vecerik, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. Sim-to-real robot learning from pixels with progressive nets. *CoRR*, abs/1610.04286, 2016. URL `http://arxiv.org/abs/1610.04286`. 2.1.3, 5.1.1

[85] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *International Conference on Machine Learning*, pages 1312–1320, 2015. 5.2.1

[86] Jürgen Schmidhuber. Powerplay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in psychology*, 4:313, 2013. 5.2.2

[87] Devin Schwab, Yifeng Zhu, and Manuela Veloso. Learning Skills for Small Size League RoboCup. In *Proceedings of the RoboCup Symposium*, Montreal, Canada, June 2018. Springer. Nominated for Best Paper Award. 1.1

[88] Devin Schwab, Yifeng Zhu, and Manuela Veloso. Zero shot transfer learning for robot soccer. In *Proceedings of AAMAS-2018, the International Conference on Autonomous Agents and Multi-Agent Systems*, Stockholm, Sweden, July 2018. Extended abstract. 1.4, 3

[89] Devin Schwab, Jost Tobias Springenberg, Murilo F. Martins, Thomas Lampe, Michael Neunert, Abbas Abdolmaleki, Tim Hertweck, Roland Hafner, Francesco Nori, and Martin A. Riedmiller. Simultaneously learning vision and feature-based control policies for real-world ball-in-a-cup. *Robotics Science and Systems (RSS)*, abs/1902.04706, 2019. URL `http://arxiv.org/abs/1902.04706`. 1.1, 2.1.3, 4

[90] Devin Schwab, Yifeng Zhu, and Manuela Veloso. Tensor action spaces for multi-agent robot transfer learning. *International Conference on Intelligent Robots and Systems (IROS) 2020*, 2020. 3

[91] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. doi: 10.1038/nature16961. URL http://dx.doi.org/10.1038/nature16961. 5.3

[92] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017. 5.3

[93] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pages 212–223. Springer, 2002. 5.1.1, 5.3.2

[94] Peter Stone and Richard S Sutton. Scaling reinforcement learning toward robocup soccer. In *Icml*, volume 1, pages 537–544. Citeseer, 2001. 2.2.2, 5.3.2

[95] Peter Stone, Richard S. Sutton, and Gregory Kuhlmann. Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005. doi: 10.1177/105971230501300301. URL http://dx.doi.org/10.1177/105971230501300301. 2.2.2, 3.6.1, 5.3, 5.3.2

[96] Peter Stone, Gregory Kuhlmann, Matthew E. Taylor, and Yaxin Liu. *Keepaway Soccer: From Machine Learning Testbed to Benchmark*, pages 93–105. RoboCup 2005: Robot Soccer World Cup IX. Springer Science + Business Media, 2006. doi: 10.1007/11780519_9. URL http://dx.doi.org/10.1007/11780519_9. 2.2.2, 5.3.2

[97] Sainbayar Sukhbaatar, arthur szlam, and Rob Fergus. Learning multiagent communication with backpropagation. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2244–2252. Curran Associates, Inc., 2016. URL http://papers.nips.cc/paper/6398-learning-multiagent-communication-with-backpropagation.pdf. 3.5, 5.1.2

[98] Richard S Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M Pilarski, Adam White, and Doina Precup. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 761–768. International Foundation for Autonomous Agents and Multiagent Systems, 2011. 5.2.1

[99] Adam Taylor, Ivana Duparic, Edgar Galván-López, Siobhán Clarke, and Vinny Cahill. Transfer learning in multi-agent systems through parallel transfer. 2013. 5.1.3

[100] Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009. 5.1.1

[101] The RoboCup Federation. RoboCup, 2019. URL http://www.robocup.org/. 1.4, 2.2.1

[102] The RoboCup Federation. Middle Size League, 2019. URL https://msl.robocup.org/. 2.2.1

[103] The RoboCup Federation. Standard Platform League, 2019. URL https://spl.robocup.org/. 2.2.1

[104] The RoboCup Federation. Small Size League, 2019. URL https://ssl.robocup.org/. 1.4, 2.2.1

[105] The RoboCup Federation. RoboCup Simulation League, 2019. URL https://www.robocup.org/leagues/23. 2.2.1

[106] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30. IEEE, 2017. 2.1.3, 5.1.1

[107] Eiji Uchibe. *Cooperative behavior acquisition by learning and evolution in a multi-agent environment for mobile robots*. PhD thesis, PhD thesis, Osaka University, 1999. 2.2.2, 5.3.2

[108] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017. 6.3

[109] Matej Vecerik, Todd Hester, Jonathan Scholz, Fumin Wang, Olivier Pietquin, Bilal Piot, Nicolas Heess, Thomas Rothörl, Thomas Lampe, and Martin A. Riedmiller. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *CoRR*, 2017. URL http://arxiv.org/abs/1707.08817. 2.2.4, 2.2.7

[110] Manuela Veloso and Peter Stone. RoboCup Robot Soccer History 1997-2011. In *Proceedings of IROS'12, the IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vilamoura, Portugal, October 2012. Extended Abstract and Video, Nominated for Best Video Award. 1.4, 2.2.1

[111] Manuela Veloso, Joydeep Biswas, Brian Coltin, and Stephanie Rosenthal. Cobots: Robust symbiotic autonomous mobile service robots. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015. 2.3.1

[112] O Vinyals, I Babuschkin, J Chung, M Mathieu, M Jaderberg, W Czarnecki, A Dudzik, A Huang, P Georgiev, R Powell, et al. Alphastar: Mastering the real-time strategy game starcraft ii. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, 2019. 5.1.2

[113] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft ii: A new challenge for reinforcement learning. *CoRR*, 2017. URL http:

//arxiv.org/abs/1708.04782v1. 3.4.2

[114] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. 2015. URL http://arxiv.org/abs/1511.06581v3. 3.6.3

[115] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989. 2.2.4

[116] Yifeng Zhu, Devin Schwab, and Manuela Veloso. Learning Primitive Skills for Mobile Robots. In *"International Conference on Robotics and Automation (ICRA)"*, Montreal, Canada, May 2019. 1.1

[117] Stefan Zickler and Manuela Veloso. Playing Creative Soccer: Randomized Behavioral Kinodynamic Planning of Robot Tactics. In *Proceedings of the RoboCup Symposium 2008*, pages 414–425, Suzhou, China, July 2008. 5.3.2

[118] Stefan Zickler and Manuela Veloso. Efficient Physics-Based Planning: Sampling Search Via Non-Deterministic Tactics and Skills. In *Proceedings of the Eighth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 27–33, Budapest, Hungary, May 2009. 5.3.2

[119] Çetin Meriçli. *Multi-Resolution Model Plus Correction Paradigm for Task and Skill Refinement on Autonomous Robots*. PhD thesis, Boğaziçi University, 2011. 5.1.1

Appendix A

Hyperparameters, Network Structures, and Other Details

This appendix include details necessary for reproduction of the experimental results discussed in the main thesis chapters. This include hyperparameters for experiments, specifics of network architectures and other details such as the automatic safety algorithms and episode reset procedures used with the real robot experiments.

A.1 RoboCup SSL Safety Checks

The action space of the robot is limited in order to keep the robots from damaging itself or the environment. The most basic safety is to reduce the maximum allowed velocity commands. This keeps the robot moving slowly, so even if a collision occurs damage will be reduced or non-existent.

The more difficult problem is keeping the robot within a predefined area without restricting exploration. While it may be fine at low speeds to let the robots bump into a physical barrier, this is less than ideal as any physical collisions have the potential to harm the robot. Instead, I have created a simple algorithm that provides "invisible walls" around an area. The agent can try any action allowed by the action space, but actions that move into the walls are cancelled out. The learning agent sees the unmodified commands, so it will need to learn that driving into the edges of the boundary do nothing and it should use other actions in its policy.

The allowed area is defined by a set of points. These points are converted to a convex hull, the edges of which are used as half-planes that divide the space into allowed and disallowed regions. Algorithm 5 describes the process of constructing the region boundaries from a set of points.

The algorithms requires a set of points $\vec{p}$ which are 2D vectors containing an x and y component. The boundary points can be easily collected by placing the robot around the allowed region and recording the points. Line 2 converts the points to a convex hull, which removes redundant points, reducing the amount of overhead while checking for boundary collisions at runtime. Line 3 gets the set of all points that form the simplex of the convex hull. In other words, the points that

Algorithm 5 Algorithm for converting boundary points to boundary regions.

1: **function** CONVERT-BOUNDARY-POINTS-TO-ALLOWED-REGIONS($\vec{p}$)
2: $C \leftarrow$ convex-hull(p)
3: $\vec{p}_S \leftarrow$ simplex-points(C)
4: $\vec{p}_c \leftarrow$ centroid(p_s)
5: $\theta_s \leftarrow [\,]$ ▷ An empty list
6: **for each** p_i in p_s **do**
7: $\vec{l} \leftarrow \vec{p}_s - \vec{p}_c$
8: $\theta_s \leftarrow \theta_s +$ atan2$(l.y, l.x)$
9: **end for**
10: $\vec{p}_s \leftarrow$ sort $\vec{p}_s$ by θ_s
11: $\vec{e}_n \leftarrow [\,]$ ▷ A list of normals of the convex hull edges
12: $e_{offset} \leftarrow [\,]$ ▷ A list of edge plane offsets
13: **for** i in [0, len(p_s)) **do**
14: $\vec{e} \leftarrow \vec{p}_s[i+1] - \vec{p}_s[i]$
15: $\vec{e}_{dir} \leftarrow \frac{\vec{e}}{||\vec{e}||}$
16: $\vec{e}_n \leftarrow \vec{e}_n + < -\vec{e}_{dir}.y, \vec{e}_{dir}.x >$
17: $\vec{e}_{midpoint} \leftarrow \vec{p}_s[i] + \frac{\vec{e}}{2}$
18: $e_{offset} \leftarrow e_{offset} + -\vec{e}_n \cdot \vec{e}_{midpoint}$
19: **end for**
20: **returns** $\vec{e_n}, e_{offset}$
21: **end function**

form the vertices of the convex hull. Line 4 computes the centroid of the hull by averaging the x and y components of all the simplex points. Lines 5 through 10 sort the simplex points so that they are counter-clockwise about the centroid. This is necessary to ensure that the edge normals computed later all have a consistent normal direction. Lines 11 through 19 iterate through pairs of the sorted points. Each pair of points forms an edge of the convex hull. This edge line is used to find the equation of a 3D plane that has a normal in the 2D plane, and contains the edge line.

The plane equation is defined as $\vec{e}_n \cdot \vec{p} + e_{offset} = d$. Each plane divides the 2D space into two halves: an allowed region and a disallowed region. Robots are allowed to move anywhere in the allowed region, but should not enter the disallowed region. If the robot is in the disallowed region, it should only be allowed to perform actions which move it towards the allowed region. By defining the regions using a plane, it is easy to check if the robot is in the allowed or disallowed region. Simply plug in the current position as $\vec{p}$ and check the sign of d. If $d < 0$ the robot is in the disallowed region, if $d > 0$ the robot is in the allowed region and if $d = 0$ the robot is on the boundary.

This leads to algorithm 6 which is run on each iteration of the training environments to modify actions sent to the radio so that the robot remains in the allowed region.

The algorithm requires a set of boundary edge planes given in $\vec{e}_n$ and e_{offset}, these come from the output of algorithm 5. The algorithm also requires the current robot position $\vec{p}_r$, the current robot orientation θ_r, the robot linear velocity command $\vec{v}_r$, the robot angular velocity command ω_r and the control time step used by the environment Δ_t. The algorithm applies the same checks

Algorithm 6 Safely modify SSL agent actions for real robot.

1: **function** MAKEROBOTACTIONSAFE($\vec{e}_n$, e_{offset}, $\vec{p}_r$, θ_r, $\vec{v}_r$, ω_r, Δ_t)
2: **for each** $(\vec{e}_{n,i}, e_{offset,i})$ in $(\vec{e}_n, e_{offset})$ **do**
3: $\vec{v}_w \leftarrow$ convert robot action $\vec{v}_r$ to world coordinates based on current orientationθ_r
4: $\vec{e}_t \leftarrow < -\vec{e}_{n,i}.y, \vec{e}_{n,i}.x >$ ▷ Compute plane tangent
5: $\vec{v}_e \leftarrow < \vec{e}_t \cdot \vec{v}_w, \vec{e}_{n,i} \cdot \vec{v}_w >$ ▷ Project velocity onto axes defined by boundary plane tangent and normal
6: $d \leftarrow \vec{e}_{n,i} \cdot \vec{p}_r + e_{offset,i}$
7: **if** $d <= 0$ **then** ▷ Robot is at boundary or in the disallowed region
8: $\vec{v}_e.x \leftarrow \max(\vec{v}_e.x, 0)$
9: **else** ▷ Robot is in the allowed region
10: **if** $\vec{v}_e.x < 0$ **then** ▷ Robot is moving towards boundary
11: $\vec{p}_{intersect} \leftarrow$ intersection point of robot with boundary given velocity command
12: $\vec{v}_{intersect} = \frac{\vec{p}_{intersect} - \vec{p}_r}{\Delta_t}$
13: $\vec{v}_e.x \leftarrow \max(\vec{v}_{intersect}.x, \vec{v}_e.x)$
14: **end if**
15: **end if**
16: $\vec{v}_w \leftarrow \vec{e}_{n,i}\vec{v}_e.x + \vec{e}_t\vec{v}_e.y$
17: $\vec{v}_r \leftarrow$ convert $\vec{v}_w$back to robot frame
18: **end for**
19: **returns** $\vec{v}_r$
20: **end function**

to each edge plane. Line 3 converts the robot action $\vec{v}_r$, which is relative to the robot's frame, into velocities in the world frame coordinates. This is necessary because the edge plane equations are with respect to the world coordinate frame. Line 4 computes the tangent of the edge plane. This allows us to express the robot action velocities in a coordinate frame defined by the edge normal and the edge plane tangent. Line 5 projects the world velocity into this coordinate frame. By performing the check in the plane's coordinate frame, it is easy to zero out components of the action that move into the disallowed region, while allowing actions that move the robot out of the disallowed region, or tangential to the boundary. Line 6 and 7 check if the robot is in the allowed region, or in the disallowed region/on the boundary. Line 8 allows actions that move the robot towards the allowed region, and prevents any movement further into the disallowed region. Line 10 checks if the robot is in the allowed region and heading for a collision with the boundary. If a collision could occur in the future, Lines 11 through 13 compute if the robot will hit in the next time step. If the robot will hit then we reduce the velocity so that it will be able to stop at the boundary, rather than driving through. Finally lines 16 through 17 reverse the transformation from edge coordinates back to the robots egocentric coordinate frame.

A.2 Chapter 2 Network and Hyperparameters

As mentioned in chapter 2 we use an actor network and critic network with fully connected layers and ReLU activation. Table A.1 shows the network specification.

Table A.1: Chapter 2 RoboCup Skill Learning Network Parameters

Network Parameter	**Value**
Actor Layer Sizes	$[300, 400]$
Critic Layer Size	$[300, 400]$
Layer Type	Fully Connected
Activation	ReLU

Table A.2 shows the hyperparameters used during training for each of the different skills. The noise function was an Ornstein-Uhlenbeck process as used in the original DDPG paper [60].

Table A.2: Chapter 2 RoboCup Skill Learning Hyperparameters

Hyperparameters	**Value**
critic learning rate	1×10^{-3}
actor learning rate	1×10^{-4}
`go-to-ball` replay mem size	1,000,000
`go-to-ball` noise parameters	$\theta = 0.15, \mu = 0, \sigma = 0.3$
`turn-and-shoot` replay mem size	450,000
`turn-and-shoot` noise parameters	$\theta = 0.15, \mu = 0, \sigma = 0.1$
`shoot-goalie` replay mem size	100,000
`shoot-goalie` noise parameters	$\theta = 0.15, \mu = 0, \sigma = 0.1$

A.3 Chapter 3 Hyperparameters and Network Architecture

The `Passing` FCQN network is the same as shown in figure 3.2 with the same layer sizes. Skip connects were used between all matching sized pairs of conv/deconv layers. All hidden layers use ReLU activations.

Below is the network structure for `Passing` using state features and fully connected layers. `c(8,4,64)` represents a convolution layer with an 8×8 filter, a stride of 4, and 64 channels.`f(512)` represents a fully connected layers whose output size is 512 hidden units.`maxpooling(3,3)` represents a max pooling layer which has `(3,3)` as the size of the pooling window. All hidden layers use ReLU activations. The network structure is: `f(256)`, `f(256)`, `f256`, and `f(`$|A|$`)`.

The `Passing` network with the tensor state input and the FC layers is: `c(12,4,32)`, `c(5,1,32)`, `maxpooling (3,3)`, `c(5,1,512)`, `flatten`, then we have the dueling layer which has a value stream as `f(2048)`, `f(1)`, and an advantage stream as `f(2048)`,

`f(|A|)`. All hidden layers use ReLU activations. `Take-the-Treasure` uses the same network architecture.

All networks were trained using the Double DQN algorithm [41] and the Adam optimizer with hyperparameters shown in Table A.3. We used ϵ-greedy exploration with epsilon decayed linearly from ϵ_{start} to ϵ_{end}.

Name	**Value**
`Passing` learning rate	1×10^{-5}
`Take-the-Treasure` learning rate	1×10^{-4}
`Breakout` learning rate	1×10^{-4}
ϵ_{start}	0.99
ϵ_{end}	0.1
ϵ decay steps	1,000,000
`Passing` replay mem size	1,000,000
`Take-the-Treasure` replay mem size	1,200,000
`Breakout` replay mem size	1,500,000

Table A.3: Hyperparameters used during training and evaluation

A.4 SSL Navigation Details

This appendix contains details related to the SSL Navigation environment described and used in chapter 4.

A.4.1 Network Structure

Table A.4 gives the details on the layer sizes for the actor and critic network used in the SSL navigation experiments. The actor network has 4 input groups: feature group, field image, robot image, goal image. The feature group uses all Fully Connected (FC) layers. The image groups start with CNN layers, the output of this is flattened and then passed to more FC layers. The output of each group is filtered using the filter lists described in section 4.3.2 and then summed together element wise. These are passed through shared FC layers and finally a separate set of FC layers for each task. Unless otherwise mentioned all layers use ReLU activations.

The critic network is similar in structure, however, the only state group input is the features. The images are not used as the asymmetric actor-critic technique is utilized. Both feature group and actions from the actor are processed by a few FC layers before being concatenated and passed through the shared FC layers.

A.4.2 Hyperparameters

Table A.5 shows the hyperparameters used in the SSL Navigation experiments from chapter 4. All experiments use the Soft Actor-Critic (SAC) training algorithm. In addition to the stochastic-

Network Section	Layer Sizes
Actor Feature Group (FC)	128
Actor Field Image CNN Filter Sizes	[8, 4, 3]
Actor Field Image CNN Strides	[4, 2, 1]
Actor Field Image FC Sizes	128
Actor Robot Image CNN Filter Sizes	[8, 4, 3]
Actor Robot Image CNN Strides	[2, 2, 1]
Actor Robot Image FC Sizes	128
Actor Goal Image CNN Filter Sizes	[8, 4, 3]
Actor Goal Image CNN Strides	[2, 2, 1]
Actor Goal Image FC Sizes	128
Actor Shared Hidden FC Sizes	[128, 128]
Actor Task Head Sizes	128
Actor Task Head Activation	Squashing Gaussian
Critic Feature Group (FC)	128
Critic Action Group (FC)	128
Critic Shared Layers (FC)	[128, 128]

Table A.4: Network structure for SSL Navigation environment.

ity from sampling the Gaussian distribution selected by the actor network, an additional amount of noise is added to the action with the probability shown in the table. The additional noise is also drawn from a Gaussian distribution with a mean of zero and the σ specified in the table.

The trainer and the sampler run in separate processes, meaning that depending on the speed of each the amount of times the same data will be sampled from the replay memory will vary. To prevent over-fitting from a slow sampler each sample can be used at most 64 times before training updates are paused to wait for new data.

A.5 Ball-in-a-Cup Details

A.5.1 Network Architecture and Hyperparameters

In this section we outline the details on the hyper-parameters used for our algorithm and baselines. Each intention policy is given by a Gaussian distribution with a diagonal covariance matrix, i.e. $\pi(\vec{a}|\vec{s},\theta) = \mathcal{N}\left(\mu, \vec{\Sigma}\right)$

The neural network outputs the mean $\mu = \mu(s)$ and diagonal Cholesky factors $A = A(s)$, such that $\Sigma = AA^T$. The diagonal factor A has positive diagonal elements enforced by the softplus transform $A_{ii} \leftarrow \log(1 + \exp(A_{ii}))$ to enforce positive definiteness of the diagonal covariance matrix.

The general network architecture we use is described in Table A.6. The image inputs are first processed by two convolutional layers followed by a fully-connected layer (see state-group heads size) followed by layer normalization. The other input modalities (features and proprioception)

Name	Value
Replay Memory Size	1,000,000
Actor Learning Rate	3×10^{-4}
Critic Learning Rate	3×10^{-4}
Actor τ	0.01
Critic τ	0.01
Gaussian Exploration Noise σ	0.2
Exploration Noise probability	0.9
Batch Size	256
Maximum Updates per Sample	64

Table A.5: Hyperparameters used during training and evaluation of the SSL Navigation environment

go through a fully connected layer (see state-group heads size), followed by layer normalization. The output of each of these three network blocks are then multiplied with 1 or 0 (depending on S^{filter}). Because all the input layers output the same shape, and we assume that the different state-groups can extract the same internal hidden representation, we sum the state-group output layers elementwise. This reduces the number of dead inputs and number of parameters needed in the first shared hidden layer. This summed output is passed to a set of fully-connected, shared hidden layers (shared layer sizes in the table). The shared hidden layer output is passed to the final output layer. In the actor, the output size is the number of means and diagonal Cholesky factors. In the critic, the output size is 1, corresponding to the Q-value of the state-action pair.

A.5.2 Real World Episode Reset Procedure

As described in the main paper we use a hand-coded reset strategy at the beginning of each episode (if the string was detected to be tangled around the arm). This procedure works as follows. Before starting the episode the robot arm is driven to a preset starting position. If the string is untangled and the ball is hanging freely, then the ball will be in the expected starting area. If the ball is not detected in this area, then the robot begins its untangle procedure. Ideally, we would know the state of the string and be able to either plan or learn a sequence of actions to untangle the string. However, the string is unobservable in features and given the low-resolution of the images, largely unobservable to the cameras. Instead, to untangle, the robot picks a random sequence of positions from a pre-set list. This list was chosen to maximize the number of twists and flips the arm does in order to maximize the chance that the string is unwrapped. After the sequence is completed the robot returns to the check position. If the ball is untangled then the next episode can begin. If the ball is still not in the expected area the entire procedure is repeated until a successful untangle.

Hyperparameters	SAC-X
2D Conv layer features (layer 1/ layer 2)	$16, 16$
2D Conv shapes (layer 1/ layer 2)	4×4, 3×3
2D Conv strides (layer 1/ layer 2)	$2 \times 2, 2 \times 2$
Actor net shared layer sizes	200, 200
Actor net state-group heads size	100
Critic net shared layer sizes	400, 400
Critic net state-group heads size	200
Discount factor (γ)	0.99
Adam learning rate	0.0001
Replay buffer size	1,000,000
Target network update period	1,000
Batch size	32
Maximum transition use	2,500
Activation function	elu
Tanh on networks input	No
Tanh on output of layer norm	Yes
Tanh on Gaussian mean	Yes
Min variance	10^{-2}
Max variance	1.0

Table A.6: Hyper parameters for SAC-X

	J0	J1	J5	J6
min. position (rad)	-0.4	0.3	0.5	2.6
max. position (rad)	0.4	0.8	1.34	4.0
min. velocity (rad/s)	-2.0	-2.0	-2.0	-2.0
max. velocity (rad/s)	2.0	2.0	2.0	2.0

Table A.7: Joint limits imposed during the experiments

A.5.3 Robot workspace

Joint limits

The Sawyer robot arm has a very large workspace[1] and can reach joint configurations which are not required to solve the Ball-in-a-Cup task or are outside of the motion capture system volume. In order to reduce the robot workspace, throughout all experiments we fixed J2, J3 and J4, and constrained position and velocity limits of the other joints, as detailed in Table A.7.

Starting position

At the beginning of every episode, the robot joints J0-J6 are commanded to $[0.0, 0.5, 0.0, -1.22, 0.0, 0.68, 3.3]$.

[1] `http://mfg.rethinkrobotics.com/intera/Sawyer_Hardware`

	J0	J1	J2	J3	J4	J5	J6
position 1	0	-2.07	0.0	0.23	0.0	0.96	0.0
position 2	0	-0.22	-2.23	0.3	0.0	1.18	3.0
position 3	0	-1.65	3.0	-0.23	0.0	0.0	3.22
position 4	0	-0.59	3.0	-0.23	0.0	0.0	3.22

Table A.8: Pre-set positions used during untangling

Untangling positions

The pre-set list of untangling positions is detailed in Table A.8.

A.5.4 Changes to Experiment Reward Functions

As mentioned in the main text, the experiment in section V-D used a set of modified reward functions. The initial real robot training used tasks 1F, 2F, 3F, 4F, 5F, 1P, 2P, 3P, 4P, and 5P. As discussed, these rewards were switched to shaped rewards to speed-up training time.

The actual tasks used in section V-D are 5F, 6F, 7F, 5P, 6P, and 7P. Tasks 1F, 2F and 3F have been replaced by shaped reward 6F. Task 4F was replaced by shaped reward 7F. Similarly, tasks 1P, 2P and 3P are replaced by shaped reward 6p. Task 4P was replaced by shaped reward 7P.