The Dissertation Committee for Matthew John Hausknecht certifies that this is the approved version of the following dissertation:

Cooperation and Communication in Multiagent Deep Reinforcement Learning

Committee:

Peter Stone, Supervisor

Dana Ballard

Ray Mooney

Risto Miikkulainen

Satinder Singh

Cooperation and Communication in Multiagent Deep Reinforcement Learning

by

Matthew John Hausknecht, B.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2016

Acknowledgments

Many thanks to my advisor Peter Stone for many years of support and encouragement throughout the PhD process. In particular, I remember a dark time when I was directionless and depressed about the prospect of finding fruitful research directions or ever graduating. You told me that all PhDs go through a period where they are lost in the woods and encouraged me to continue onward. This thesis is testament to the fact that given sufficient persistence, encouragement, and caffeine, there is a way out of the woods.

Thanks to my lab-mates and colleagues who have shaped my research and added flavor to my graduate experience: Shivaram Kalyanakrishnan, Juhyun Lee, Todd Hester, Samuel Barrett, Daniel Urieli, Piyush Khandelwal, Patrick MacAlpine, Katie Genter, Karl Pichotta, Wesley Tansey, Subhashini Venugopalan, Jake Menashe, Elad Liebman, Sanmit Narvekar, Josiah Hanna, Matteo Leonetti, and many others.

Additional thanks to my committee: Dana Ballard, Ray Mooney, Risto Miikkulainen, and Satinder Singh for the time and feedback they so generously gave.

Thanks to Kay Nettle and Amy Bush for tirelessly supporting and debugging the many issues that arise from setting up and maintaining a GPU cluster. The research in this thesis would have been impossible (or at least computationally intractable) without your help.

Special thanks to my loving family and my fiancé Man Liang for their unconditional support throughout the ups and downs of the PhD process.

Cooperation and Communication in Multiagent Deep Reinforcement Learning

Matthew John Hausknecht, Ph.D.
The University of Texas at Austin, 2016

Supervisor: Peter Stone

Reinforcement learning is the area of machine learning concerned with learning which actions to execute in an unknown environment in order to maximize cumulative reward. As agents begin to perform tasks of genuine interest to humans, they will be faced with environments too complex for humans to predetermine the correct actions using hand-designed solutions. Instead, capable learning agents will be necessary to tackle complex real-world domains. However, traditional reinforcement learning algorithms have difficulty with domains featuring 1) high-dimensional continuous state spaces, for example pixels from a camera image, 2) high-dimensional parameterized-continuous action spaces, 3) partial observability, and 4) multiple independent learning agents. We hypothesize that deep neural networks hold the key to scaling reinforcement learning towards complex tasks. This thesis seeks to answer the following two-part question:

1) How can the power of Deep Neural Networks be leveraged to extend Reinforcement Learning to complex environments featuring partial observability, high-dimensional parameterized-continuous state and action spaces, and sparse rewards? 2) How can multiple Deep Reinforcement Learning agents learn to cooperate in a multiagent setting?

To address the first part of this question, this thesis explores the idea of using recurrent neural networks to combat partial observability experienced by agents in the domain of Atari 2600 video games. Next, we design a deep reinforcement learning agent capable of discovering effective policies for the parameterized-continuous action space found in the Half Field Offense simulated soccer domain.

To address the second part of this question, this thesis investigates architectures and algorithms suited for cooperative multiagent learning. We demonstrate that sharing parameters and memories between deep reinforcement learning agents fosters policy similarity, which can result in cooperative behavior. Additionally, we hypothesize that communication can further aid cooperation, and we present the Grounded Semantic Network (GSN), which learns a communication protocol grounded in the observation space and reward function of the task. In general, we find that the GSN is effective on domains featuring partial observability and asymmetric information.

All in all, this thesis demonstrates that reinforcement learning combined with deep neural network function approximation can produce algorithms capable of discovering effective policies for domains with partial observability, parameterized-continuous actions spaces, and sparse rewards. Additionally, we demonstrate that single agent deep reinforcement learning algorithms can be naturally extended towards cooperative multiagent tasks featuring learned communication. These results represent a non-trivial step towards extending agent-based AI towards complex environments.

Table of Contents

Chapter 1
Introduction

One of the core challenges of artificial intelligence is designing generally capable learning agents: an ideal learning agent would be capable of learning in a diversity of situations or environments without requiring extensive redesign or reprogramming for each new problem it encounters. In particular, it is undesirable for the developer of the agent to have to understand the environment in which the agent will be deployed or have to optimize the sensory inputs or motor outputs of the agent to provide a greater chance of successful learning. Instead, we hope that through machine learning, we might develop intelligent agents capable of learning from noisy, raw, or unprocessed input signals, acting through low-level motor primitives, and reach high performance on a diverse set of tasks.

However, many questions remain about how to design agents capable of fulfilling these criteria. Broadly, *How should an agent be represented?* and *What computations should an intelligent agent perform?* There have been answers to these questions developed throughout the history of AI research. This thesis adopts a particular perspective on these questions through the use of deep neural networks to represent learning agents and reinforcement learning to govern the computations performed by the agent. These choices are by no means the only options for developing intelligent agents, but are currently considered promising approaches. Building on this foundation, this thesis contributes several novel designs for artificially intelligent agents capable of learning to handle low-level, noisy, or incomplete sensations; act using low-level motor primitives; and cooperate with other learning agents. The prior advances in representation and computation that made this thesis possible are further discussed below.

Over the last several years, deep neural networks have shown strong performance on a variety of supervised learning tasks, and are now considered state-of-the-art general-purpose function approximators for the tasks of image recognition (Szegedy et al., 2014), speech recognition (Hinton et al., 2012), translation

(Sutskever et al., 2014), and text generation (Graves, 2013). The success of deep learning has paralleled the growth and availability of large, labeled datasets such as ImageNet (Russakovsky et al., 2015). These large datasets along with the advent of affordable and highly parallel graphics processing unit based computing allow models with an extremely large number of parameters to be efficiently trained and optimized.

However, many problems of interest lack supervised labels and well-defined datasets. Of particular interest to this thesis are problems involving autonomous agents interacting with unknown environments. For this reason, successful approaches for combining deep neural networks with reinforcement learning have taken more time to mature. There are particularly notable exceptions: the Deep Q-Network (DQN) (Mnih et al., 2015) demonstrated super-human performance across a variety of different Atari 2600 video games. More recently, reinforcement learning augmented with deep neural network function approximation and Monte Carlo tree search was used to create a champion level Go player called Alpha Go (Silver et al., 2016).

At a high level, reinforcement learning (RL) is an area of machine learning concerned with how agents ought to take actions in an environment so as to maximize cumulative reward. Broadly, reinforcement learning answers the question of *what computations an intelligent agent should perform.* More specifically, reinforcement learning addresses the problem of sequential decision making in an unknown environment. The environment provides a loosely supervised signal in the form of a reward after each decision made by the agent. The goal of the agent is to maximize the cumulative reward accrued throughout the course of interacting with the environment. Because the agent is attempting to maximize cumulative reward, individual rewards do not necessarily indicate correct decisions. Thus rewards are less informative than supervised labels and are not directly usable as targets when training deep neural networks.

Nonetheless, the results presented in this thesis demonstrate that it is possible to harness the power of deep neural networks for the purpose of reinforcement learning. In particular, deep RL agents are capable of learning in higher dimen-

sional, more complicated environments than possible before the advent of deep networks, environments featuring partial observability, high dimensional raw-pixel input, and parameterized-continuous action spaces.

In the real world, learning often happens in groups rather than all alone. From the time of infancy nearly all animals learn from their parents or other members of the species. From this perspective, single agent learning is limited in the sense that the learning agent is alone and can only use its own experiences with the environment as guidance. We suspect that multiple learning agents, working together, can accomplish far more than a single agent ever could. To this end, a major focus of this thesis is on exploring architectures and approaches for cooperative multiagent deep RL. In particular, we examine the sharing of neural network parameters, the sharing of replay memories, and the use of learned communication protocols.

1.1 Research Question

Motivated by the design of capable general-purpose learning agents, we pose the following questions:

Thesis Question: 1) How can the power of Deep Neural Networks be leveraged to extend Reinforcement Learning towards domains featuring partial observability, continuous parameterized action spaces, and sparse rewards? **2)** How can multiple Deep Reinforcement Learning agents learn to cooperate in a multiagent setting?

To better organize our work on this question, we presented individual contributions of the thesis in the next section.

1.2 Contributions

In fulfillment of the thesis question we present the following contributions:

1. **Deep Recurrent Q-Network (DRQN): An exploration of recurrency as a method of dealing with partial observability**: Recurrency allows neural networks to process inputs sequentially through time. We develop a recurrent-neural-network-based controller called DRQN and show that in certain Atari games, recurrent processing of inputs can be quite beneficial, particularly when the inputs are observations generated from a noisy or partially observable environment. DRQN is presented in full detail in Chapter 3.
2. **Half-Field-Offense: An open-source environment for developing and testing cooperative learning agents**: The Half Field Offense (HFO) domain is built on the RoboCup 2D soccer simulator and features a parameterized-continuous action space in which discrete actions must paired with continuous parameters. HFO supports single agent, ad-hoc, and multiagent learning. This is the domain for exploring partial observability, parameterized action space, and cooperative multiagent learning. Half-Field-Offense is presented in Section 2.5.
3. **Deep RL in parameterized action space**: We tackle the full reinforcement learning problem in the Half Field Offense domain. Specifically, we extend an existing algorithm for for deep reinforcement learning in continuous action space, making several modifications for improved learning in HFO's parameterized action space. Chapter 4 discusses deep RL in HFO and provides details on the techniques necessary for fast and stable learning.
4. **An exploration of multiagent Deep RL in HFO**: We explore cooperative multiagent learning in the HFO domain. Specifically, we demonstrate how parameter and memory sharing architectures can promote cooperative learning between deep reinforcement learning agent. Finally, we propose an experiment to examine if two agents, given the optional ability to communicate with each other, will learn to use language as part of coordinating towards a common goal. We examine several different approaches for using learned communication to achieve better cooperation. This contribution is spread across Chapters 5 and 6.
5. **Curriculum Learning in HFO**: We present a method to enable learning

agents to perform well on tasks that have highly sparse rewards. Specifically, we break down the target task into a curriculum of subtasks, learn each subtask, and use the resulting skills to learn the target task. Tasks are represented to the agent using an embedding space, in which the network's activations are altered as a function of the active task. We demonstrate in Chapter 7 that this architecture allows a single network to maintain high performance across multiple tasks.

While the resulting agents are not quite ready for RoboCup competition, taken together we believe these contributions represent a nontrivial step towards addressing the thesis question. In the next section, we present a high-level overview of the thesis.

1.3 Dissertation Overview

The remainder of this thesis is organized as follows: **Chapter 2** presents background on Markov Decision Processes (MDPs), Reinforcement Learning (RL), Deep Learning, and the Atari Environment (ALE) and Half-Field-Offense environemnt (HFO). These two environments are used throughout the remainder of the thesis. Additionally, Chapter 2 describes two existing deep reinforcement learning algorithms, DQN and DDPG. These algorithms form the basis of the learning agents in later chapters. **Chapter 3** presents the first technical contribution, a deep reinforcement learning architecture designed to combat partial observability in the input space of a learning agent. Next, **Chapter 4** presents an extension of Deep RL into parameterized action space and demonstrates successful single agent learning in Half-Field-Offense. Building on these single agent results, **Chapter 5** examines approaches for cooperative multiagent deep reinforcement learning. In particular this chapter describes sharing gradients, experiences, and weights between the agents. Examining another aspect of cooperation, **Chapter 6** demonstrates that agents can learn to communicate with each other in order to enhance cooperation. **Chapter 7** presents a curriculum learning approach for decomposing complex tasks with sparse rewards into sequences of easier tasks. Finally, related work can be

found in Chapter 8. Chapter 9 discusses future work and concludes.

It is possible to understand much of the work in this thesis without having to read it cover to cover. The related work in Chapter 8 can be understood on its own. However, the majority of the other chapters assume basic knowledge of reinforcement learning and neural networks, presented in Chapter 2. A brief review of this chapter is recommend before attempting to understand later chapters.

Beyond the background, Chapter 3 on recurrent neural networks for partial observability may also be understood on its own. The multiagent architectures in Chapters 5-7 build on the architecture for single agent learning in parameterized action space presented in Chapter 4. These chapters can also be understood at a high level without knowing the specifics of the single agent architecture.

Figure 1.1 depicts the dependencies between the chapters. Finally, while ever acronym is at least defined the first time it is used, there is a table of abbreviations located at the end of the thesis which may prove helpful if an unknown acronym is encountered.

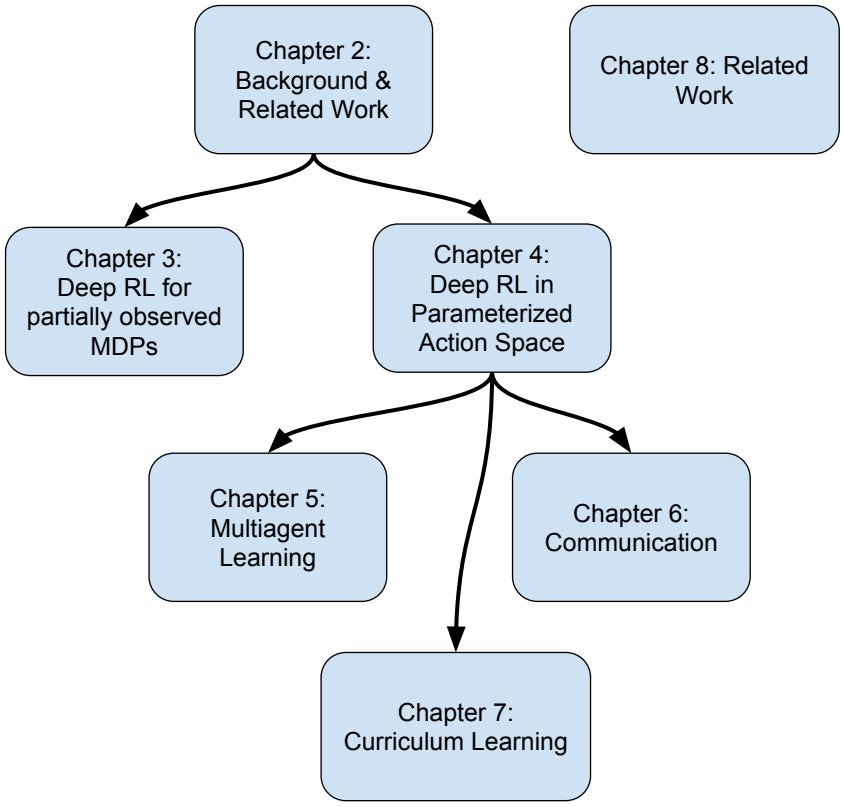


Figure 1.1: Arrows denote dependencies between thesis chapters. Chapters 5-7 build on the architecture and techniques presented in Chapter 4, but can also be understood at a high level on their own.

Chapter 2
Background

This chapter presents the theoretical background underlying the algorithms and concepts presented in later chapters. This chapter is designed to be understood alone or as a precursor to the following chapters. We first present the Markov Decision Process, or MDP, in short. The MDP is a theoretical foundation upon which reinforcement learning is grounded. The next section discusses reinforcement learning at a high level and presents two common RL algorithms: Q-Learning and SARSA. Subsequently, we refocus the discussion on deep learning and present the basics of deep neural networks. Meaningfully combining reinforcement learning with deep learning is a major focus of this thesis and we review the Deep Q-Network (DQN) and Deep Deterministic Policy Gradients (DDPG) algorithms for deep reinforcement learning in discrete and continuous action spaces.

2.1 Markov Decision Processes

The Markov Decision Process, or MDP, is a common paradigm for modeling sequential decision making problems, problems in which learning agents must make repeated choices about what actions to perform and how to deal with the consequences of those choices. In this thesis we will focus on MDPs in which time is modeled as a sequence of discrete units called timesteps. At each timestep the agents follow a fixed pattern of interaction with the MDP: first the agents perceive the state of the world s, next each agent selects some action a to perform. Once all agents have selected their actions, the state of the world advances to the next timestep and the agents are each given a reward r indicating how well they are performing the task. This cycle of perception and action continues until the end of the episode. Episodes end when the goal has been achieved or some other termination condition is met.

Formally an MDP is a 5-tuple (S, A, P, R, γ). S is a set possible states.

Each timestep the agent perceives the current state of the environment $s \in S$. A is a set of possible actions available to the agent. The agent must choose which action $a \in A$ to select at each timestep. We assume the set of available actions does not change as a function of time. $P : S \times A \times S \rightarrow [0, 1]$ is a transition function which determines how the environment advances after actions have been selected. Specifically, the environment provides a new state $s' \sim P(s, a)$. $R : S \times A \rightarrow \mathbb{R}$ is a reward function that provides the agent with a scalar reward at each timestep. Finally γ is a scalar that specifies how much priority should be given to immediate versus long-term rewards.

2.2 Reinforcement Learning

Broadly, reinforcement learning is concerned with choosing a sequence of actions that maximizes cumulative reward. A reinforcement learning agent must learn a policy π which is a function that maps from environment states $s \in S$ to actions $a \in A$. The learned policy is good if the sum of individual rewards accrued throughout all timesteps of the episode is large. This discounted sum of accrued rewards is referred to as the return $J = \sum_{t=0}^{\infty} \gamma^t r_t$.

The challenge of reinforcement learning is to find an optimal policy π^* which maximizes return. One common way of approaching this challenge is to break the problem of learning a sequence of actions into many smaller problems: that of learning the correct action to take at each state. Instead of directly learning the best action to select, a common approach is to instead estimate expected sum of long term rewards that will be accrued from taking a particular action in a particular state. Such an estimate is referred to as a value function because it approximates the sum of rewards or value.

This thesis will focus on learning Q-Values (also called action-values). A Q-Value $Q^\pi(s, a)$ is a scalar that estimates the expected sum of gamma-discounted rewards that will be accrued by taking action a in state s and following policy π thereafter. Q-Value estimates are iteratively refined by visiting the same s, a pair in multiple episodes and averaging the resulting returns, or by using the Bellman

equation (Bellman, 1957) to estimate the Q-Value from the immediate reward and the Q-Values of the next state and action:

$$Q(s, a) = R(s, a) + \gamma Q(s', a')$$

This equation forms the basis of an algorithm called SARSA (Sutton and Barto, 1998), in which Q-Values are iteratively refined by using experience tuples (s, a, r, s') gathered from interacting with the environment. The SARSA update is a type of dynamic programming because the cached estimate of $Q(s', a')$ is being used to refine the estimate of $Q(s, a)$. By visiting a wide variety of state-action pairs and iterating this update, Q-Value estimates will provably converge to the correct estimates of long term reward $Q^*(s, a)$. Using Q-Value estimates, it is possible to recover a policy by selecting the action with the maximal Q-Value from the current state. In this manner, a correct Q-Value function Q^* directly produces an optimal policy π^*.

Q-Learning (Watkins and Dayan, 1992) is another effective algorithm for learning Q-Values. In contrast to SARSA, Q-Learning takes a max over all next state actions:

$$Q(s, a) = R(s, a) + \gamma \max_{a'} Q(s', a')$$

By using a max, Q-Learning updates the Q-Value towards the best action, possibly not the one that was actually selected. Such an update allows the agent to select potentially bad exploratory actions, without having devasting effects on the Q-Values. This distinction is known as on-policy vs. off-policy learning. SARSA is on-policy because updates to Q-Values are performed directly on the actions taken by the current policy. Q-Learning is off-policy since updates employ a max over all possible actions, potentially updating with an action that was not actually taken by the current policy.

Subsequent chapters will reinterpret the SARSA and Q-Learning updates in the context of deep neural network. We next introduce neural networks and deep learning.

2.3 Deep Neural Networks

Deep neural networks are parametric models composed of multiple stacked layers of artificial neurons. They are accepted to be powerful general function approximators. In short, a deep network computes a nested function of the form:

$$\hat{y} = f_k(W_k \dots f_1(W_1 x + b_1) \cdots + b_k)$$

Input example x is transformed through k network layers into output prediction $\hat{y}$. Each layer transforms its input, first multiplying by a weight matrix W_i, adding on a vector of biases b_i, and finally applying a nonlinear transform f_i. The final layer of the network represents its prediction $\hat{y}$. By comparing the prediction $\hat{y}$ to the correct label y, it is possible to determine if the network is performing correctly and to train it to more closely approximate the target.

Specifically, neural networks are trained using the Backpropagation algorithm (Werbos, 1974). Backpropagation seeks to minimize a loss function, typically the difference between the prediction and the correct label: $L = |\hat{y} - y|$. Backpropagation provides gradients $\nabla_\theta(L)$ which specify how the parameters of the network should be changed in order to reduce the loss. Throughout this thesis we will use theta to denote the concatenation of all the parameters in the model $\theta = (W_1, b_1, W_2, b_2, \dots, W_k, b_k)$.

The choice of nonlinear activation function greatly affects the depth and training time of a neural network. The classic logistic sigmoid activation function takes the form: $f(x) = 1/(1 + e^{-x})$. This function was used to train neural networks for many years, but recently has been supplanted by the Rectified Linear Unit (ReLU). ReLU units have an activation of the form $f(x) = \max(0, x)$ and have been shown to result in much faster training of deep neural networks. Unless otherwise noted, ReLU activation functions are used in every layer of every network throughout this thesis.

Advances in deep learning also came in the form of more advanced optimization methods. Backpropagation provides gradients which indicate directions in parameter space of immediate change. These gradients may be used in vanilla

stochastic gradient descent (SGD). One important improvement on SGD was momentum. The innovation was to remember these gradients after each iteration of backpropagation rather than discarding them. By maintaining a history of gradients, it's possible to more quickly escape certain types of local optima by building momentum in the direction of consistent gradients and removing momentum in the directions of fluctuating gradients.

Further improvement came in the form of adaptive learning rate optimization methods which maintain a per-parameter learning rate. Three examples of adaptive learning rate methods are RMSProp (Tieleman and Hinton, 2012), AdaDelta (Zeiler, 2012), and Adam (Kingma and Ba, 2014). Adaptive learning rate methods proved to be quite useful for reinforcement learning because of the changing nature of the data. For example, consider an agent that inhabits a certain region of the state space for an extended duration and then discovers a new region of the state space. Under a fixed decaying learning rate, the agent's learning rate may be too small to effectively learn about the new region of state space it has discovered. With an adaptive learning rate method, the agent's learning rate can be automatically adjusted to reflect the new batch of experiences. Throughout this thesis we use the Adam optimizer and the ReLU nonlinearity.

2.3.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) (LeCun et al., 1998) were a key innovation for learning from images. At a high level, CNNs are a method of sharing weights across pixel space. Instead of learning a full weight matrix, a CNN instead learns low-dimensional filters which are convolved across the input pixels. These filters have been shown to learn hierarchical detectors for edges, image parts, and full objects. CNNs have proven highly successful at static image recognition problems such as the MNIST, CIFAR, and ImageNet Large-Scale Visual Recognition Challenge (Krizhevsky et al., 2012; Szegedy et al., 2014; Zeiler and Fergus, 2014). By using a hierarchy of trainable filters and feature pooling operations, CNNs are capable of automatically learning complex features required for visual object recognition tasks achieving superior performance to hand-crafted features. Throughout

this thesis, we will employ convolutional layers when states are represented by images. Otherwise, standard fully-connected layers will be preferred.

Having introduced reinforcement learning and deep learning, we next present the two main domains that will be used as testbeds in this thesis: the Arcade Learning Environment (ALE) and Half-Field-Offense (HFO).

2.4 Arcade Learning Environment

The Arcade Learning Environment (Bellemare et al., 2013) is an evaluation platform in which agents can interact with Atari 2600 games. Each of the 61 supported games is its own environment and presents unique learning challenges to the agent. Common across all games is the state, action, and reward representations. These common representations make it feasible to evaluate a single learning agent on many different tasks. In the remainder of this section, we review the state, action, and reward representations for ALE.

2.4.1 ALE: State Space

The most natural state representation for Atari games is the game screen. Game screens are 160×210 pixels in size and feature standard 3-channel RGB. Processing a representation consisting of pixels requires a powerful and efficient function approximator. Convolutional neural networks (Section 2.3.1) provide just such a tool. Notably the pixel representation is exactly what is observed by a human when playing the game. The work presented in Chapter 3 employs a preprocessed version of this screen representation.

Of note, ALE also provides access to the RAM state of the Atari console. The Atari 2600 had only 128 bytes (1024 bits) of RAM. While small, these RAM bits are sufficient to completely restore the state of any Atari game.

2.4.2 ALE: Action Space

The action space of ALE consists of 18 discrete actions. To understand these actions it is helpful to understand the physical controller: the Atari 2600 uses a joystick that is capable of moving in the eight directions (North, South, East, West) as well as combinations (North-East, South-West, etc), or not moving at all. Thus, there are a total of 9 possible joystick positions. Additionally, there is a single button. The Atari 2600 console registers combinations of the possible joystick states with or without a button press: the 18 actions correspond to the 9 possible joystick locations when the button is depressed and the 9 possible joystick locations when the button is unperturbed.

Some games use only a subset of the 18 possible actions. For example, in the game of Pong, only joystick movements in the North-South direction matter. In other games, certain actions are illegal: in the game of Skiing, pressing the fire button resets the player to the beginning of the course, and including it as a possible action would result in learning agents that could reset the MDP. Fortunately, ALE provides information for each game about which actions are legal, and which are actually used by each game. We employ this information to reduce the size of the action space for certain games.

Figure 2.1: The physical Atari 2600 features 18 discrete actions defined by the positions of the joystick and button. Games are typically represented using the 160×210 dimensional RGB screen representation. The picture on the right shows an example game of Space Invaders.

2.4.3 ALE: Rewards

Rewards in ALE are largely based on accrued game score. By keeping track of which RAM bits are used to encode the game score, ALE can determine the current score of the game and provide a reward at each timestep equal to the change in game score. One remaining difficulty is that the magnitude of reward varies from game to game, making it difficult to directly assess the performance of a learning agent. This has been addressed in the past by normalizing against a reference score (Bellemare et al., 2013), comparing to human performance (Mnih et al., 2015), or using Z-Scores (Hausknecht et al., 2013).

ALE is used at the primary environment in Chapter 3. The subsequent chapters use the Half Field Offense domain, which we present now.

2.5 Half Field Offense Domain

RoboCup is an international robot soccer competition that promotes research in AI and robotics. Within RoboCup, the *2D simulation league* works with an abstraction of soccer wherein the players, the ball, and the field are all 2-dimensional objects. However, for the researcher looking to quickly prototype and evaluate different algorithms, the full soccer task presents a cumbersome prospect: full games are lengthy, have high variance in their outcome, and demand specialized handling of rules such as free kicks and offsides.

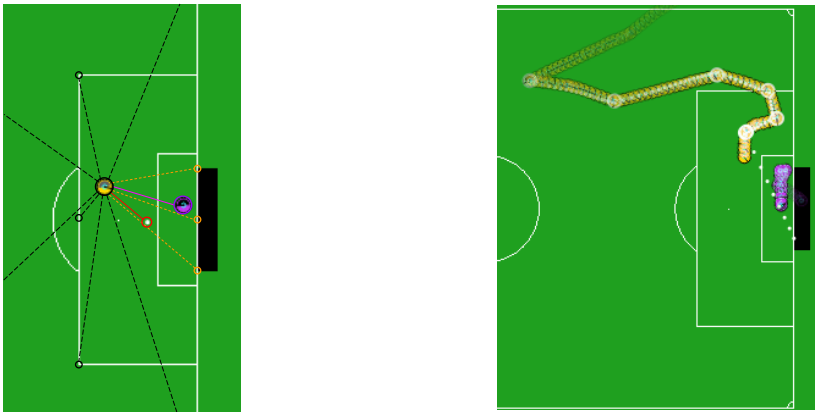
The Half Field Offense domain[1] abstracts away the difficulties of full RoboCup and exposes the experimenter only to core decision-making logic, and to focus on the most challenging part of a RoboCup 2D game: scoring and defending goals. In HFO, each agent receives its own state sensations and must independently select its own actions. HFO is naturally characterized as an episodic multi-agent POMDP because of the sequential partial observations and actions on the part of the agents and the well-defined episodes which culminate in either a goal being scored or the ball leaving the play area. To begin each episode, the agent and ball are positioned

[1]Example videos of Half Field Offense games may be viewed at: Video: `https://vid.me/sNev` Video: `https://vid.me/JQTw` Video: `https://vid.me/1b5D`.

randomly on the offensive half of the field. The episode ends when a goal is scored, the ball leaves the field, or 500 timesteps pass. The following subsections introduce the low-level state and action space used by agents in this domain.

2.5.1 State Space

The agent uses a low-level, egocentric viewpoint encoded using 58 continuously-valued features. These features are derived through Helios-Agent2D's (Akiyama, 2010) world model and provide angles and distances to various on-field objects of importance such as the ball, the goal, and the other players. Figure 2.2 depicts the perceptions of the agent. The most relevant features include: Agent's position, velocity, orientation, and stamina; Indicator if the agent is able to kick; Angles and distances to the following objects: Ball, Goal, Field-Corners, Penalty-Box-Corners, Teammates, and Opponents. A full list of state features may be found at `https://github.com/mhauskn/HFO/blob/master/doc/manual.pdf`.



(a) State Space (b) Helios Champion

Figure 2.2: **Left:** HFO State Representation uses a low-level, egocentric viewpoint providing features such as distances and angles to objects of interest like the ball, goal posts, corners of the field, and opponents. **Right:** Helios handcoded policy scores on a goalie. This 2012 champion agent forms a natural (albeit difficult) baseline for comparison.

2.5.2 Action Space

Half Field Offense features a low-level, parameterized action space. There are four mutually-exclusive discrete actions: Dash, Turn, Tackle, and Kick. At each timestep the agent must select one of these four to execute. Each action has 1-2 continuously-valued parameters which must also be specified. An agent must select both the discrete action it wishes to execute as well as the continuously valued parameters required by that action. The full set of parameterized actions is:

- `Dash`(power, direction): Moves in the indicated direction with a scalar power in $[0, 100]$. Movement is faster forward than sideways or backwards.
- `Turn`(direction): Turns to indicated direction.
- `Tackle`(direction): Contests the ball by moving in the indicated direction. This action is only useful when playing against an opponent.
- `Kick`(power, direction): Kicks the ball in the indicated direction with a scalar power in $[0, 100]$. All directions are parameterized in the range of $[-180, 180]$ degrees.

2.5.3 Teammates

Automated teammates and opponents in HFO use a policy derived from Helios, the 2012 RoboCup 2D champion team (Akiyama, 2010). This policy is designed for full 11-versus-11 matches, but gracefully scales to any of the smaller tasks in the HFO umbrella. As our benchmark results indicate, automated teammates and opponents using the Helios policy exhibit strong but not perfect policies. More importantly, Helios teammates favor cooperation and will strategically pass the ball to player-controlled agents. While some passes are direct, lead passes require the player-agent to quickly reposition in order to receive. When the player has the ball, Helios teammates intelligently position themselves and will sprint to receive a pass from the player.

2.5.4 Evaluation Metrics

Having presented the basic state spaces, action spaces, and NPCs featured in the HFO Environment, we now address the important question of how to evaluate the performance of HFO agents.

The HFO environment does not provide reward signals and instead indicates the ending status of the game. HFO episodes end with one of the following termination conditions:

Goal: The offense scored a goal.
Captured (CAP): The defense gained control of the ball.
Out of Bounds (OOB): The ball left the play field.
Out of Time (OOT): No agent has approached the ball in the last 100 timesteps.

Using these termination conditions, we propose two evaluation metrics: Goal Percentage and Time to Goal. The primary focus of learning in HFO is to score goals when playing offense and prevent goals from being scored when playing defense. The primary metric, **Goal Percentage**, the percentage of all trials that end with a goal being scored, captures exactly this notion. The hallmark of an effective offensive agent is a high goal percentage. A second metric, **Time to Goal** (TTG), is defined as the number of timesteps required to score in each trial that culminates with a goal. Efficient offensive agents typically seek to minimize time to goal, while defenders strive to maximize this metric.

2.5.5 Learning Paradigms

The HFO Environment supports several learning paradigms: **Single Agent Learning**, involves a lone offensive or defensive agent playing against one or many opponents. In **Ad Hoc Teamwork**, the agent must learn to cooperate with one or more unknown teammates without pre-coordinated strategies (Barrett, 2014; Stone et al., 2010). In the case of HFO, learning agents have the opportunity to act as the ad hoc teammate of the Helios agents. Finally, **Multiagent Learning** (see Chapter 5) places two or more learning agents on the same team with the shared objective of scoring or defending the goal. Known as Multiagent Reinforcement Learn-

ing (MARL), the challenge for these agents is to learn both individual competency as well as cooperation (Tan, 1998). While not examined in this thesis, HFO also supports configurations that blend these learning paradigms. For example, a team could consist of several learning agents paired with one or more Helios teammates, mixing multiagent learning with ad hoc teamwork. Additionally, HFO can create multiagent scenarios in which agents have competing objectives, for example by allocating some learning agents to play offense and others to play defense.

2.5.6 Related Work: RoboCup Soccer

Much of the work in this thesis takes place within the RoboCup 2D Half Field Offense domain. RoboCup 2D soccer has a rich history of learning, which is reviewed in this section.

In one of the earliest examples, Andre and Teller (1999) used Genetic Programming to evolve policies for RoboCup 2D Soccer. By using a sequence of reward functions, they first encourage the players to approach the ball, kick the ball, score a goal, and finally to win the game. Similarly, our work features players whose policies are entirely trained and have no hand-coded components. Our work differs by using a gradient-based learning method and learning from demonstration rather than a reward signal.

Competitive RoboCup agents are primarily hand-coded but may feature components that are learned or optimized for better performance. Examples include the Brainstormers who used neural reinforcement learning to optimize individual skills such as intercepting and kicking the ball (Riedmiller and Gabel, 2007). However, these skills were optimized in the context of a larger, already working policy. Similarly, MacAlpine et al. (2015a) employed the layered-learning framework to incrementally learn a series of interdependent behaviors. Such learning techniques have been shown to be applicable to physical robots in addition to simulated ones (Kohl and Stone, 2004; Hausknecht and Stone, 2010; da Silva et al., 2014). Instead of optimizing small portions of a larger policy, we take the approach of learning the full policy.

Another promising approach to learning in RoboCup 2D used planning in

conjunction with MAX-Q hierarchical value function decomposition of the action space to drive learning (Bai et al., 2012, 2013). This framework has been implemented by the WrightEagle team, winners of the 2013 and 2014 RoboCup 2D competitions.

Recently, Barrett completed a thesis on Ad-Hoc teamwork and demonstrated the continued relevance of HFO for exploring novel learning algorithms (Barrett, 2014; Barrett and Stone, 2015).

Masson and Konidaras explored reinforcement learning in parameterized action spaces and formalized the concept of a Parameterized Action Markov Decision Process (PAMDP) (Masson and Konidaris, 2015). Using a handcrafted HFO-like domain, they learn a policy for scoring on a keeper.

There is much more work that uses the 2D RoboCup domain. The work in this thesis differs from the above approaches by focusing on Deep RL as the learning method for RoboCup 2D. Additionally, this thesis is not targeted at producing a competitive RoboCup team. Instead, we choose to focus on exploring different the limits of current deep reinforcement learning approaches without the burden of producing competition-ready agents. The next section describes two existing deep reinforcement learning algorithms.

2.6 Deep Q-Network (DQN)

The amount of work combining deep learning with reinforcement learning has expanded in the last few years. This section reviews the DQN algorithm for learning in discrete action spaces.

Deep Q-Learning (Mnih et al., 2015) extends standard Q-Learning (Watkins and Dayan, 1992) by using a deep neural network as a Q-Value function approximator. For a review of Q-Learning see Section 2.2. Challenging reinforcement learning domains such as Atari games feature far too many unique states to maintain a separate estimate for each state-action. Instead a model is used to approximate the Q-values and generalize between similar state-actions. Generalization allows sensible Q-Value estimates to be derived for state-actions that have never been en-

countered before. Additionally, the large number of parameters in deep networks allow performance and accuracy to scale with large amounts of experience data.

In the case of Deep Q-Learning, the model is a neural network parameterized by weights and biases collectively denoted as θ. Q-values are estimated online by querying the output nodes of the network after performing a forward pass given a state input. Q-values estimated by the neural network are denoted $Q(s, a|\theta)$.

Updates are made to the parameters of the network to minimize the following differentiable loss function:

$$L(s, a|\theta_i) = \left(r + \gamma \max_{a'} Q(s', a'|\theta_i) - Q(s, a|\theta_i)\right)^2 \tag{2.1}$$

$$\theta_{i+1} = \theta_i + \alpha \nabla_\theta L(\theta_i) \tag{2.2}$$

Since $|\theta| \ll |S \times A|$, the neural network model naturally generalizes beyond the states and actions it has been trained on. However, because the same network is generating the next state target Q-values that are used in updating its current Q-values, such updates can oscillate or diverge (Tsitsiklis and Roy, 1997). Deep Q-Learning uses two techniques to restore learning stability:

First, experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ are recorded in a replay queue $\mathcal{D}$. When performing updates, experiences are drawn at random from this queue. This selection process decorrelates the agent's current state with the states used in the updates. Such a step is necessary in order to ensure the pool of experience that is being drawn on for updates is not overly biased by the preferences of the current policy. More recent studies have shed additional light on the utility of the replay memory, as follows.

First, the A3C algorithm (Mnih et al., 2016) showed that with sixteen copies of the same agent running in parallel environments, a replay memory is no longer necessary. Instead, it suffices to simply update from the most recent experiences encountered by the different agents. Because the agents likely occupy different parts of the state space, the diversity of experience is sufficient to obviate the need for a replay queue. Second, in multiagent settings, Foerster et al. (2016a) show that the

replay memory didn't help with learning. Finally, Schaul et al. (2015) showed that prioritizing experiences in the replay memory could help DQN's performance on Atari games, in contrast to the standard method of selecting experiences at random.

In addition to the replay memory, the second main improvement was a separate, **target network** Q^- that provides update targets to the main network. Q^- is identical to the main network except its parameters θ^- are updated to match θ at a slow frequency. The advantage of the target is that is decouples the feedback resulting from the network generating its own targets. So far, target networks have remained crucial for deep reinforcement learning.

Incorporating these improvements, we will now revisit the update equation. Specifically, at each training iteration i, an experience $e_t = (s_t, a_t, r_t, s_{t+1})$ is sampled uniformly from the replay memory $\mathcal{D}$. The loss of the network is determined as follows:

$$L_i(\theta_i) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim\mathcal{D}}\left[\Big(y_i - Q(s_t, a_t; \theta_i)\Big)^2\right] \tag{2.3}$$

where $y_i = r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'; \theta^-)$ is the update target given by the target network $\hat{Q}$. Updates performed in this manner have been empirically shown to be tractable and stable (Mnih et al., 2015).

Deep, model-free RL in discrete action spaces can be performed using the Deep Q-Learning method (Mnih et al., 2015) which employs a single deep network to estimate the value function of each discrete action and, when acting, selects the maximally valued output for a given state input. Several variants of DQN have been explored including decaying traces (Narasimhan et al., 2015), double Q-Learning (van Hasselt et al., 2015), and dueling networks (Wang et al., 2015). These networks work well in continuous state spaces but do not function in continuous action spaces because the output nodes of the network, while continuous, are trained to output Q-Value estimates rather than continuous actions. The next section will present an algorithm for handling continuous action spaces.

2.7 Continuous Action Space: DDPG

Extending deep RL to continuous action spaces, Lillicrap et al. (2015) introduced the Deep Deterministic Policy Gradient (DDPG) algorithm. Based on the Actor/Critic architecture (Sutton and Barto, 1998), this algorithm maintains separate networks: for the actor and the critic. The critic network is updated using temporal difference learning, like in DQN. However, the actor's policy is updated from the gradients suggested by the critic network.

An Actor/Critic architecture (Sutton and Barto, 1998) provides one solution to this problem by decoupling the value learning and the action selection. Represented using two deep neural networks, the actor network outputs continuous actions while the critic estimates the value function. The actor network μ, parameterized by θ^{μ}, takes as input a state s and outputs a continuous action a. The critic network Q, parameterized by θ^{Q}, takes as input a state s and action a and outputs a scalar Q-Value $Q(s, a)$. Figure 2.3 shows Critic and Actor networks.

Updates to the critic network are largely unchanged from the standard temporal difference update used originally in Q-Learning (Watkins and Dayan, 1992) and later by DQN:

$$Q(s, a) = Q(s, a) + \alpha\big(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\big) \tag{2.4}$$

Adapting this equation to the neural network setting described above results in minimizing a loss function defined as follows:

$$L_Q(s, a|\theta^Q) = \Big(Q(s, a|\theta^Q) - \big(r + \gamma \max_{a'} Q(s', a'|\theta^Q)\big)\Big)^2 \tag{2.5}$$

However, in continuous action spaces, this equation is no longer tractable as it involves maximizing over next-state actions a'. Instead we ask the actor network to provide a next-state action $a' = \mu(s'|\theta^{\mu})$. This yields a critic loss with the

following form:

$$L_Q(s, a|\theta^Q) = \Big(Q(s, a|\theta^Q) - \big(r + \gamma Q(s', \mu(s'|\theta^\mu)'|\theta^Q)\big)\Big)^2 \tag{2.6}$$

The value function of the critic can be learned by gradient descent on this loss function with respect to θ^Q. However, the accuracy of this value function is highly influenced by the quality of the actor's policy, since the actor determines the next-state action a' in the update target.

The critic's knowledge of action values is then harnessed to learn a better policy for the actor. Given a sample state, the goal of the actor is to minimize the difference between its current output a and the optimal action in that state a^*.

$$L_\mu(s|\theta^\mu) = \big(a - a^*\big)^2 = \big(\mu(s|\theta^Q) - a^*\big)^2 \tag{2.7}$$

The critic may be used to provide estimates of the quality of different actions but naively estimating a^* would involve maximizing the critic's output over all possible actions: $a^* \approx \arg\max_a Q(s, a|\theta^Q)$. Instead of seeking a global maximum, the critic network can provide gradients which indicate directions of change, in action space, that lead to higher estimated Q-Values: $\nabla_a Q(s, a|\theta^Q)$. To obtain these gradients requires a single backward pass over the critic network, much faster than solving an optimization problem in continuous action space. Note that these gradients are not the common gradients with respect to parameters. Instead these are gradients with respect to inputs, first used in this way by NFQCA (Hafner and Riedmiller, 2011). To update the actor network, these gradients are placed at the actor's output layer (in lieu of targets) and then back-propagated through the network. For a given state, the actor is run forward to produce an action that the critic evaluates, and the resulting gradients may be used to update the actor:

$$\nabla_{\theta^\mu}\mu(s) = \nabla_a Q(s, a|\theta^Q)\nabla_{\theta^\mu}\mu(s|\theta^\mu) \tag{2.8}$$

Alternatively one may think of these updates as simply interlinking the actor

and critic networks: On the forward pass, the actor's output is passed forward into the critic and evaluated. Next, the estimated Q-Value is backpropagated through the critic, producing gradients $\nabla_a Q$ that indicate how the action should change in order to increase the Q-Value. On the backwards pass, these gradients flow from the critic through the actor. An update is then performed only over the actor's parameters. Figure 2.3 shows an example of this update.

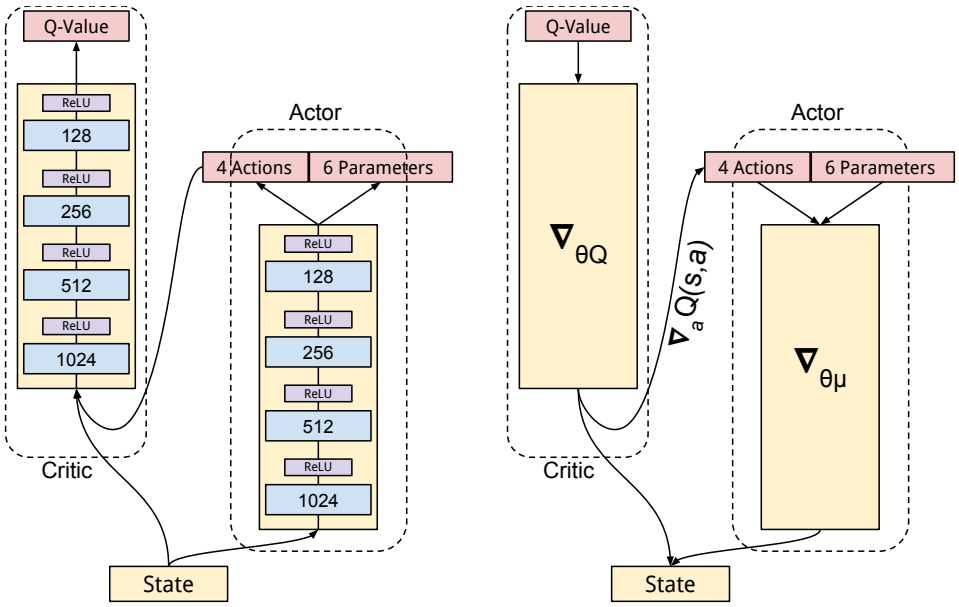


Figure 2.3: **Actor-Critic architecture (left)**: actor and critic networks may be interlinked, allowing activations to flow forwards from the actor to the critic and gradients to flow backwards from the critic to the actor. The gradients coming from the critic indicate directions of improvement in the continuous action space and are used to train the actor network without explicit targets. **Actor Update (right)**: Backwards pass generates critic gradients $\nabla_a Q(s, a|\theta^Q)$ w.r.t. the action. These gradients are back-propagated through the actor resulting in gradients w.r.t. parameters ∇_{θ^μ} which are used to update the actor. Critic gradients w.r.t. parameters ∇_{θ^Q} are ignored during the actor update.

2.7.1 Stable Updates

Updates to the critic rely on the assumption that the actor's policy is a good proxy for the optimal policy. Updates to the actor rest on the assumption that the critic's gradients, or suggested directions for policy improvement, are valid when tested in the environment. It should come as no surprise that several techniques are necessary to make this learning process stable and convergent.

Because the critic's policy $Q(s, a|\theta^Q)$ influences both the actor and critic updates, errors in the critic's policy can create destructive feedback resulting in divergence of the actor, critic, or both. To resolve this problem (Mnih et al., 2015) introduce a Target-Q-Network Q', a replica of the critic network that changes on a slower time scale than the critic. This target network is used to generate next state targets for the critic update (Equation 2.6). Similarly a Target-Actor-Network μ' combats quick changes in the actor's policy.

The second stabilizing influence is a replay memory $\mathcal{D}$, a FIFO queue consisting of the agent's latest experiences (typically one million). Updating on experiences sampled uniformly from this memory reduces bias compared to updating exclusively from the most recent experiences.

Employing these two techniques the critic loss in Equation 2.6 and actor update in Equation 2.7 can be stably re-expressed as follows:

$$L_Q(\theta^Q) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim\mathcal{D}}\left[\Big(Q(s_t, a_t) - \big(r_t + \gamma Q'(s_{t+1}, \mu'(s_{t+1}))\big)\Big)^2\right] \tag{2.9}$$

$$\nabla_{\theta^\mu}\mu = \mathbb{E}_{s_t\sim\mathcal{D}}\Big[\nabla_a Q(s_t, a|\theta^Q)\nabla_{\theta^\mu}\mu(s_t)|_{a=\mu(s_t)}\Big] \tag{2.10}$$

Finally, these updates are applied to the respective networks, where α is a per-parameter step size determined by the gradient descent algorithm. Additionally, the target-actor and target-critic networks are updated to smoothly track the actor

and critic using a factor $\tau \ll 1$:

$$\begin{aligned}
\theta^Q &= \theta^Q + \alpha \nabla_{\theta^Q} L_Q(\theta^Q) \\
\theta^\mu &= \theta^\mu + \alpha \nabla_{\theta^\mu} \mu \\
\theta^{Q'} &= \tau \theta^Q + (1-\tau)\theta^{Q'} \\
\theta^{\mu'} &= \tau \theta^\mu + (1-\tau)\theta^{\mu'}
\end{aligned} \tag{2.11}$$

One final component is an adaptive learning rate method such as ADADELTA (Zeiler, 2012), RMSPROP (Tieleman and Hinton, 2012), or ADAM (Kingma and Ba, 2014).

2.8 Chapter Summary

This chapter presented background on Markov Decision Processes (MDPs), the formalism underlying all of reinforcement learning. Additionally, the basic notions of states, actions, and rewards will be used throughout this thesis. Reinforcement learning provides a methodology for learning how to select actions in MDPs in such a way as to maximize long term discounted rewards. We presented value functions which estimate the sum of rewards resulting from taking a specific action from a specific state. In order to learn a value function we reviewed the SARSA and Q-Learning algorithms which use bootstrapping to estimate the value of a particular state-action from the immediate reward and value of the next state-action. Finally, this chapter presented background on deep neural networks, including discussion of convolutional neural networks, nonlinear activation functions such as ReLU, and solvers such as ADAM. Together these form the building blocks for the algorithms presented in the chapters that follow.

Additionally, this chapter covered the Atari and Half-Field-Offense domains. At a high level, the Atari domain fits well with the goals of robust domain-indepedent learning because it contains so many diverse environments to interact with. An agent that could learn effective control policies on the full set of Atari games is

far more convincing than an agent that can learn only on a single game. However, the Atari domain is not without limitations: first, Atari games only feature discrete actions, a simplifying assumption when compared to many real world and robotics-based domains that require continuous control. Second, the vast majority of Atari games are limited to single agent. The Half-Field-Offense domain addresses these shortcomings. It features a parameterized-continuous action space and the ability to learn with multiple agents. However, unlike Atari, the feature space of HFO is not image-based and thus does not require convolutional neural network processing. Additionally, while HFO features a diversity of different tasks, they all fall under the heading of soccer. The set of different Atari games is broader and more varied. Taken together, these domains represent a diverse set of tasks that spans visual image and feature based perception, discrete and continuous action spaces, and single and multiagent learning. Chapter 3 presents further work in the Atari domain and chapters 4-7 pertain to HFO.

Finally, this chapter presented two methods for combining deep neural networks with reinforcement learning. The first method, Deep Q-Learning uses a deep neural network as a function approximator to learn a critic - an estimator for the Q-Value of each discrete action. Temporal difference updates allow the network to bootstrap targets from a history of experiences stored in a replay memory. A target network is used to improve stability of learning.

In continuous action space, the DDPG algorithm uses an actor-critic architecture where both actor and critic networks are approximated using deep neural networks. Unlike DQN, the critic network takes an action as input and outputs a Q-Value. The critic is trained using temporal difference updates and the actor is trained by backpropagation - following the gradients that maximize the critic's estimated Q-Value. To ensure stability during learning, both the actor and critic networks use target networks. Due to the difficulty of finding a max in continuous action space, off-policy updates are approximated by using an action generated by actor's current policy.

Chapter 3
Deep RL for Partially Observed MDPs

Chapter 2 introduced the basics of deep learning and reinforcement learning then touched on some specific algorithms for deep reinforcement learning including DQN and DDPG. This chapter describes using recurrent neural networks to extend DQN to help handle partial observability. To understand this chapter, a solid understanding of Atari domain (presented in Chapter 2.4), MDPs and partial observability (Chapter 2.1) as well as the DQN algorithm is required. For additional background on DQN, see Chapter 2.6. This chapter addresses the portion of the thesis question regarding how deep neural networks can be leveraged to extend reinforcement learning towards domains featuring partial observability.[1] This chapter is the basis of thesis contribution 1.[2]

Deep Q-Networks (DQNs) have been shown to be capable of learning human-level control policies on a variety of different Atari 2600 games (Mnih et al., 2015). True to their name, DQNs learn to estimate the Q-Values (or long-term discounted returns) of selecting each possible action from the current game state. Given that the network's Q-Value estimate is sufficiently accurate, a game may be played by selecting the action with the maximal Q-Value at each timestep. Learning policies mapping from raw screen pixels to actions, these networks have been shown to achieve state-of-the-art performance on many Atari 2600 games.

However, Deep Q-Networks are limited in the sense that they learn a mapping from a limited number of past states, or game screens in the case of Atari 2600. In practice, DQN is trained using an input consisting of the last four states the agent has encountered. Thus DQN will be unable to master games that require the player to remember events more distant than four screens in the past. Put differently, any game that requires a memory of more than four frames will appear non-Markovian because the future game states (and rewards) depend on more than

[1] This Chapter is based in part on the following publication (Hausknecht and Stone, 2015).
[2] See list of thesis contributions in Chapter 1.2.

just DQN's current input. Instead of a Markov Decision Process (MDP), the game becomes a Partially-Observable Markov Decision Process (POMDP).

Real-world tasks often feature incomplete and noisy state information resulting from partial observability. As Figure 3.1 shows, given only a single game screen, many Atari 2600 games are POMDPs. One example is the game of Pong in which the current screen only reveals the location of the paddles and the ball, but not the velocity of the ball. Knowing the direction of travel of the ball is a crucial component for determining the best paddle location.

(a) Pong (b) Frostbite (c) Double Dunk

Figure 3.1: Nearly all Atari 2600 games feature moving objects. Given only one frame of input, Pong, Frostbite, and Double Dunk are all POMDPs because a single observation does not reveal the velocity of the ball (Pong, Double Dunk) or the velocity of the icebergs (Frostbite).

We observe that DQN's performance declines when given incomplete state observations and hypothesize that DQN may be modified to better deal with POMDPs by leveraging advances in Recurrent Neural Networks. Therefore we introduce the *Deep Recurrent Q-Network* (DRQN), a combination of a Long Short Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) and a Deep Q-Network. Crucially, we demonstrate that DRQN is capable of handing partial observability, and that when trained with full observations and evaluated with partial observations, DRQN better handles the loss of information than does DQN. Thus, recurrency confers

benefits as the quality of observations degrades. This result is the basis of the thesis contribution 1 regarding leveraging deep neural networks towards domains that feature partial observability.

3.1 Partial Observability

In real world environments it is rare that the full state of the system can be provided to the agent or even be determined. In other words, the Markov property rarely holds in real world environments. A Partially Observable Markov Decision Process (POMDP) better captures the dynamics of many real-world environments by explicitly acknowledging that the sensations received by the agent are only partial glimpses of the underlying system state. Formally a POMDP can be described as a 6-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \Omega, \mathcal{O})$. $\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}$ are the states, actions, transitions, and rewards as before, except now the agent is no longer privy to the true system state and instead receives an observation $o \in \Omega$. This observation is generated from the underlying system state according to the probability distribution $o \sim \mathcal{O}(s)$. Deep Q-Learning, as originally described (Mnih et al., 2015), has no explicit mechanisms for deciphering the underlying state of the POMDP and is only effective if the observations are reflective of underlying system states. In the general case, estimating a Q-value from an observation can be arbitrarily bad since $Q(o, a|\theta) \neq Q(s, a|\theta)$.

Our experiments show that adding recurrency to Deep Q-Learning allows the Q-network network to better estimate the underlying system state, narrowing the gap between $Q(o, a|\theta)$ and $Q(s, a|\theta)$. Stated differently, recurrent deep Q-networks can better approximate actual Q-values from sequences of observations, leading to better policies in partially observed environments.

3.2 DRQN Architecture

To isolate the effects of recurrency, we minimally modify the architecture of DQN, replacing only its first fully connected layer with a recurrent LSTM layer of the same size. These changes are minimal in the sense that no new layers are added

and the number of nodes in each layer is unchanged. Alternative architectures are presented in Section 3.10. Depicted in Figure 3.2, the architecture of DRQN takes a single 84×84 preprocessed image. Specifically, the first hidden layer convolves 32 8×8 filters with stride 4 across the input image and applies a rectifier nonlinearity. The second hidden layer convolves 64 4×4 filters with stride 2, again followed by a rectifier nonlinearity. The third hidden layer convolves 64 3×3 filters with stride 1, followed by a rectifier. The outputs from this convolutional stack are fed to the fully connected LSTM layer (Hochreiter and Schmidhuber, 1997). Finally, a linear layer outputs a Q-Value for each action. During training, the parameters for both the convolutional and recurrent portions of the network are learned jointly from scratch.

3.3 Stable Recurrent Updates

Updating a recurrent, convolutional network requires each backward pass to contain many time-steps of game screens and target values. Additionally, the LSTM's initial hidden state may either be zeroed or carried forward from its previous values. We consider two types of updates:

Bootstrapped Sequential Updates: Episodes are selected randomly from the replay memory and updates begin at the beginning of the episode and proceed forward through time to the conclusion of the episode. The targets at each timestep are generated from the target Q-network, $\hat{Q}$. The RNN's hidden state is carried forward throughout the episode.

Bootstrapped Random Updates: Episodes are selected randomly from the replay memory and updates begin at random points in the episode and proceed for only *unroll iterations* timesteps (e.g. one backward call). The targets at each timestep are generated from the target Q-network, $\hat{Q}$. The RNN's initial state is zeroed at the start of the update.

Algorithm 1 provides pseudocode for both types of update. Sequential updates have the advantage of carrying the LSTM's hidden state forward from the beginning of the episode. However, by sampling experiences sequentially for a full

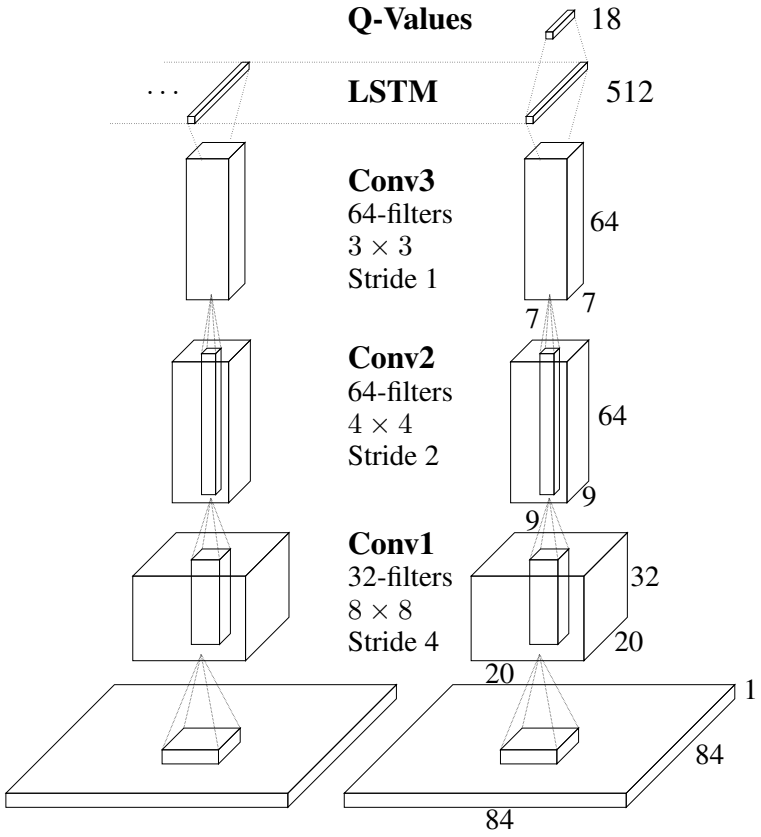


Figure 3.2: DRQN convolves three times over a single-channel image of the game screen. The resulting activations are processed through time by an LSTM layer. The last two timesteps are shown here. LSTM outputs become Q-Values after passing through a fully-connected layer. Convolutional filters are depicted by rectangular sub-boxes with pointed tops.

episode, they violate DQN's random sampling policy.

Random updates better adhere to the policy of randomly sampling experience, but, as a consequence, the LSTM's hidden state must be zeroed at the start of each update. Zeroing the hidden state makes it harder for the LSTM to learn functions that span longer time scales than the number of timesteps reached by back propagation through time.

Informal experiments comparing these two updates indicate that both types of updates are viable and yield convergent policies with similar performance across

Algorithm 1 Recurrent Update Algorithms

procedure BOOTSTRAPPED SEQUENTIAL UPDATE
    sample episode E randomly from $\mathcal{D}$
    $h \leftarrow < 0, \ldots, 0 >$ ▷ Initial LSTM hidden state is all zeroes.
    $n \leftarrow 0$ ▷ Start update a timestep zero.
    for timestep $\{t_n \ldots t_{n+unroll}\} \in E$ **do**
        $h \leftarrow$ BPTT$(\{t_n \ldots t_{n+unroll}\}, h)$ ▷ Backpropagation through time.
        $n \leftarrow n + unroll$ ▷ Unroll is 10 in experiments.

procedure BOOTSTRAPPED RANDOM UPDATE
    sample episode E randomly from $\mathcal{D}$
    sample n randomly from $\{0, \ldots, |E| - unroll\}$
    $h \leftarrow < 0, \ldots, 0 >$
    timestep $\{t_n \ldots t_{n+unroll}\} \in E$
    BPTT$(\{t_n \ldots t_{n+unroll}\}, h)$

a set of games. Therefore, to limit complexity, all results herein use the randomized update strategy. We expect that all presented results would generalize to the case of sequential updates.

Having addressed the architecture and updating of a Deep Recurrent Q-Network, we now show how it performs on domains featuring partial observability.

3.4 Atari Games: MDP or POMDP?

The state of an Atari 2600 game is fully described by the 128 bytes of console RAM. Humans and agents, however, observe only the console-generated game screens. For many games, a single game screen is insufficient to determine the state of the system. DQN infers the full state of an Atari game by expanding the state representation to encompass the last four game screens. Many games that were previously POMDPs now become MDPs. Of the 49 games investigated by (Mnih et al., 2015), the authors were unable to identify any that were partially observable given the last four frames of input.[3] Since the explored games are fully observable

[3]Some Atari games are undoubtedly POMDPs such as Blackjack in which the dealer's cards are hidden from view. Unfortunately, Blackjack is not supported by the ALE emulator.

given four input frames, we need a way to introduce partial observability without reducing the number of input frames given to DQN.

3.5 Flickering Pong POMDP

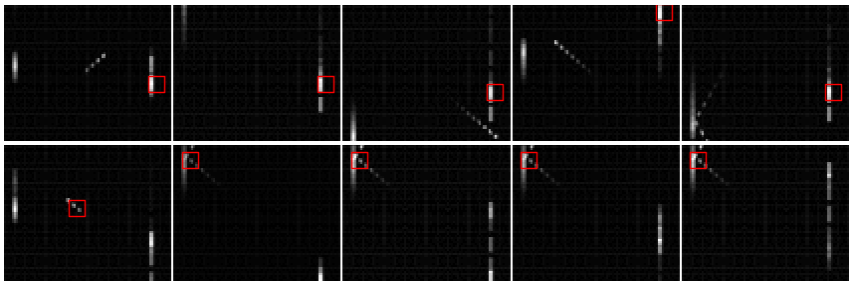
To address this problem, we introduce the *Flickering Pong* POMDP - a modification to the classic game of Pong such that at each timestep, the screen is either fully revealed or fully obscured with probability $p = 0.5$. Obscuring frames in this manner probabilistically induces an incomplete memory of observations needed for Pong to become a POMDP.

In order to succeed at the game of Flickering Pong, it is necessary to integrate information across frames to estimate relevant variables such as the location and velocity of the ball and the location of the paddle. Since half of the frames are obscured in expectation, a successful player must be robust to the possibility of several potentially contiguous obscured inputs.

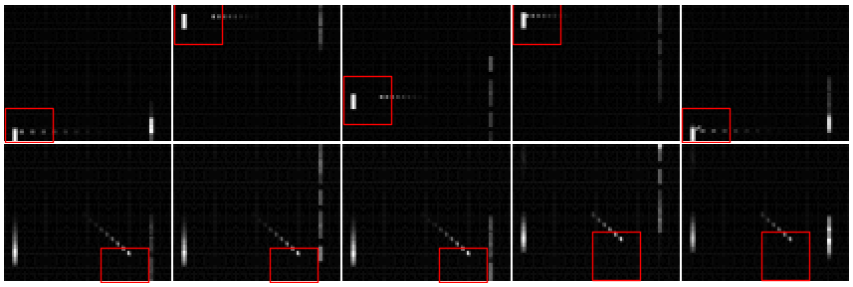
We train 3 types of networks to play Flickering Pong: the recurrent 1-frame DRQN, a standard 4-frame DQN, and an augmented 10-frame DQN. The standard 4-frame DQN is trained in order to compare directly to previous work (Mnih et al., 2015). The 1-frame DRQN is trained using BPTT for the last 10-frames, so it has an effective memory of length 10. To make the comparison fair, we trained a version of DQN that also has access to the same amount of history - e.g. 10 frames. Thus, both 10-frame DQN and 1-frame DRQN have access to the same length of history and only differ in their architecture. As Figure 3.5 indicates, providing more frames to DQN improves performance. Nevertheless, even with 10 frames of history, DQN still struggles to achieve positive scores.

Perhaps the most important opportunity presented by a history of game screens is the ability to convolutionally detect object velocity. Figure 3.3 visualizes the game screens maximizing the activations of different convolutional filters and confirms that the 10-frame DQN's filters do detect object velocity, though perhaps less reliably than normal unobscured Pong.[4]

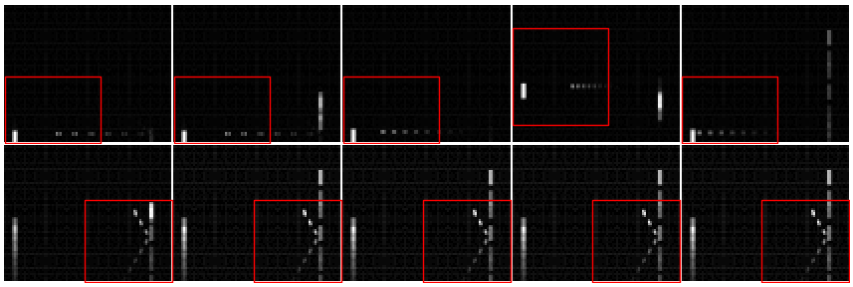
[4](Guo et al., 2014) also confirms that convolutional filters learn to respond to patterns of move-



(a) Conv1 Filters

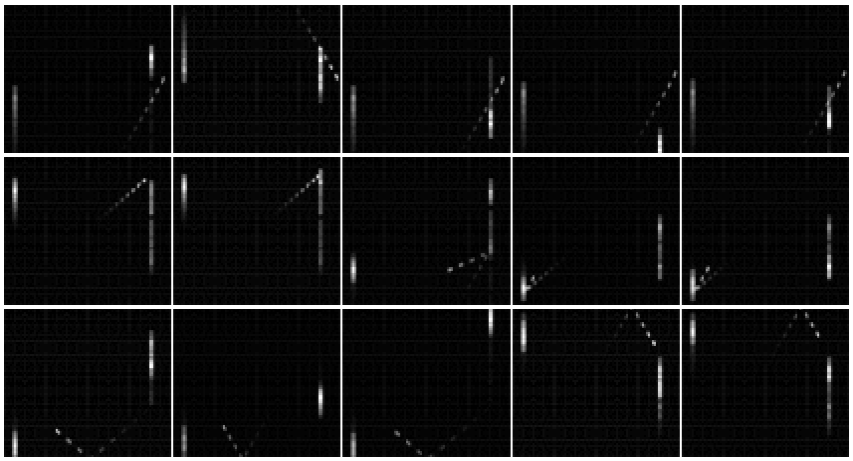


(b) Conv2 Filters



(c) Conv3 Filters

Figure 3.3: Convolution filters learned by 10-frame DQN on the game of Pong. Each row plots the input frames that trigger maximal activation of a particular convolutional filter in the specified layer. The red bounding box illustrates the portion of the input image that caused the maximal activation. Most filters in the first convolutional layer detect only the paddle. Conv2 filters begin to detect ball movement in particular directions and some jointly track the ball and the paddle. Nearly all Conv3 filters track ball and paddle interactions including deflections, ball velocity, and direction of travel.



(a) Image sequences maximizing LSTM units

Figure 3.4: Frame sequences that maximally activate DRQN's LSTM units. Each row corresponds to a single LSTM unit (out of DRQN's 512 LSTM units) and each column depicts the 10-frame sequences that maximally activate that unit. In other words, each row shows the five in-game scenarios that caused a specific LSTM unit to maximally respond. The three LSTM units shown were selected because they exhibited interesting and highly-interpretable patterns of behavior. Not all 512 LSTM units were as interpretable, but many were. In general, despite seeing only a single frame at a time, individual LSTM units also detect high level events: the agent missing the ball, ball reflections off of paddles, and ball reflections off the walls.

Remarkably, DRQN performs well at this task even when given only one input frame per timestep. With a single frame it is impossible for DRQN's convolutional layers to detect any type of velocity. Instead, the higher-level recurrent layer must compensate for both the flickering game screen and the lack of convolutional velocity detection. Even so, DRQN regularly achieves scores exceeding 10 points out of a maximum of 21. Figure 3.4 confirms that individual units in the LSTM layer are capable of integrating noisy single-frame information through time to detect high-level Pong events such as the player missing the ball, the ball reflecting on a paddle, or the ball reflecting off the wall.

DRQN is trained using backpropagation through time for the last ten timesteps. Thus both the non-recurrent 10-frame DQN and the recurrent 1-frame DRQN have access to the same history of game screens.[5] DRQN makes better use of the limited history to achieve higher scores.

Thus, when dealing with partial observability, a choice exists between using a non-recurrent deep network with a long history of observations or using a recurrent network trained with a single observation at each timestep. Flickering Pong provides an example in which a recurrent deep network performs better even when given access to the same number of past observations as the non-recurrent network. The performance of DRQN and DQN is further compared across as set of ten games (Table 3.1), where no systematic advantage is observed for either algorithm.

3.6 Experimental Details

In all experiments reported in the remainder of this chapter, DRQN and DQN policies were evaluated every 50,000 iterations by playing 10 episodes and averaging the resulting scores. Networks were trained for 10 million iterations and used a replay memory of size 400,000. Additionally, all networks used ADADELTA (Zeiler, 2012) optimizer with a learning rate of 0.1 and momentum of 0.95. LSTM's gradients were clipped to a value of 10 to ensure learning stability. All other settings

ment seen in game objects.

[5]However, (Karpathy et al., 2015) show that LSTMs can learn functions at training time over a limited set of timesteps and then generalize them at test time to longer sequences.

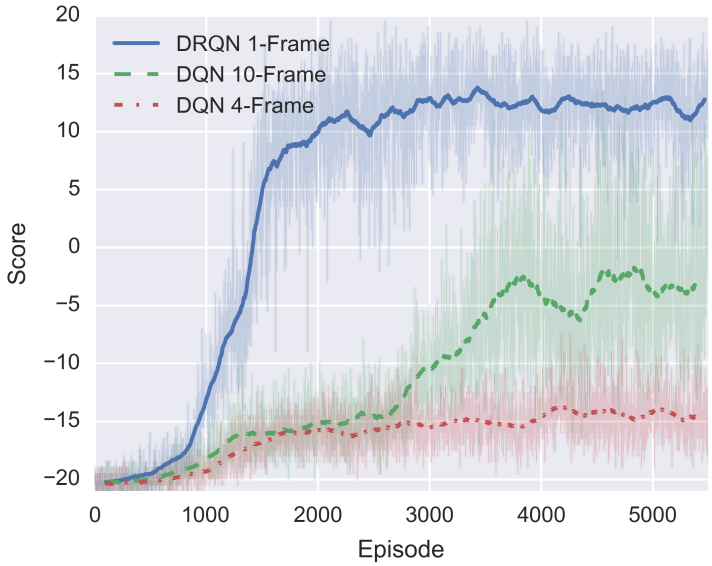


Figure 3.5: Flickering Pong

Figure 3.6: In the partially-observable *Flickering Pong* environment, DRQN proves far more capable at handling the noisy sensations than DQN despite having only a single frame of input. Lacking recurrency, 4-frame DQN struggles to overcome the partial observability induced by the flickering game screen. This struggle is partially, but not entirely, remedied by providing DQN with 10-frames of history.

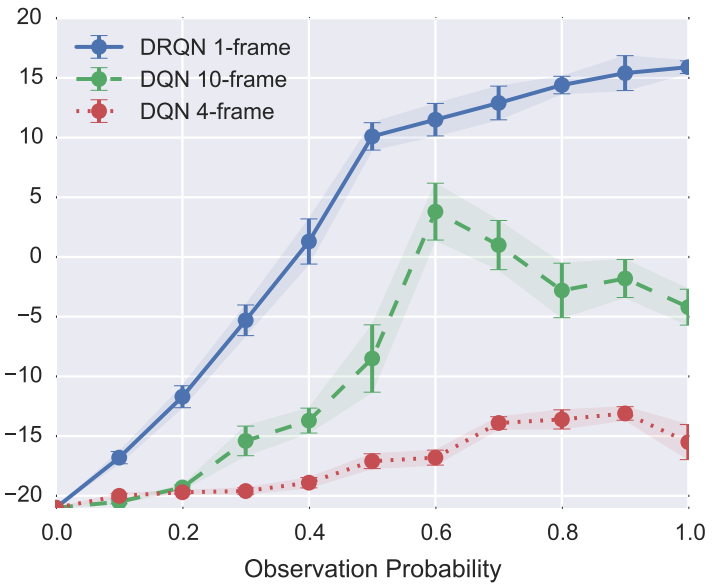


Figure 3.7: Policy Generalization

Figure 3.8: After being trained with observation probability of 0.5, the learned policies are then tested for generalization. The policies learned by DRQN generalize gracefully to different observation probabilities. DQN's performance peaks slightly above the probability on which it was trained. Errorbars denote standard error.

were identical to those given in (Mnih et al., 2015). All networks were trained using the Arcade Learning Environment ALE (Bellemare et al., 2013). The following ALE options were used: color averaging, minimal action set, and death detection. DRQN is implemented in Caffe (Jia et al., 2014).

3.7 Generalization Performance

To analyze the generalization performance of the Flickering Pong players, we evaluate the best policies for DRQN, 10-frame DQN, and 4-frame DQN while varying the probability of obscuring the screen. Note that these policies were all trained on Flickering Pong with $p = 0.5$ and are now evaluated against different p values. Figure 3.7 shows that DRQN performance continues improving as the probability of observing a frame increases. In contrast, 10-frame DQN's performance peaks near the observation probability for which it has been trained and declines even as more frames are observed. Thus DRQN learns a policy which allows performance to scale as a function of observation quality. Such a property is valuable for domains in which the quality of observations varies through time.

3.8 Evaluation on Standard Atari Games

We selected the following nine Atari games for evaluation: *Asteroids* and *Double Dunk* feature naturally-flickering sprites making them good potential candidates for recurrent learning. *Beam Rider*, *Centipede*, and *Chopper Command* are shooters. *Frostbite* is a platformer similar to Frogger. *Ice Hockey* and *Double Dunk* are sports games that require positioning players, passing and shooting the puck/ball, and require the player to be capable of both offense and defense. *Bowling* requires actions to be taken at a specific time in order to guide the ball. *Ms Pacman* features flickering ghosts and power pills. These nine games span different genres (shooting, exploration, sports, etc) and were selected to be a representative subset of the full set of Atari games.

Given the last four frames of input, all of these games are MDPs rather than

POMDPs. Thus there is no reason to expect DRQN to outperform DQN. Indeed, results in Table 3.1 indicate that on average, DRQN does roughly as well DQN. Specifically, our re-implementation of DQN performs similarly to the original, outperforming the original on five out of the nine games, but achieving less than half the original score on Centipede and Chopper Command. DRQN performs outperforms our DQN on the games of Frostbite and Double Dunk, but does significantly worse on the game of Beam Rider (Figure 3.9). The game of Frostbite (Figure 3.1b) requires the player to jump across all four rows of moving icebergs and return to the top of the screen. After traversing the icebergs several times, enough ice has been collected to build an igloo at the top right of the screen. Subsequently the player can enter the igloo to advance to the next level. As shown in Figure 3.9, after 12,000 episodes DRQN discovers a policy that allows it to reliably advance past the first level of Frostbite.

	DRQN $\pm std$	DQN $\pm std$	
Game		Ours	Mnih et al.
Asteroids	1020 ($\pm$312)	1070 ($\pm$345)	1629 ($\pm$542)
Beam Rider	3269 ($\pm$1167)	**6923** ($\pm$1027)	6846 ($\pm$1619)
Bowling	62 ($\pm$5.9)	72 ($\pm$11)	42 ($\pm$88)
Centipede	3534 ($\pm$1601)	3653 ($\pm$1903)	8309 ($\pm$5237)
Chopper Cmd	2070 ($\pm$875)	1460 ($\pm$976)	6687 ($\pm$2916)
Double Dunk	**-2** ($\pm$7.8)	-10 ($\pm$3.5)	-18.1 ($\pm$2.6)
Frostbite	**2875** ($\pm$535)	519 ($\pm$363)	328.3 ($\pm$250.5)
Ice Hockey	-4.4 ($\pm$1.6)	-3.5 ($\pm$3.5)	-1.6 ($\pm$2.5)
Ms. Pacman	2048 ($\pm$653)	2363 ($\pm$735)	2311 ($\pm$525)

Table 3.1: On standard Atari games, DRQN performance parallels DQN, excelling in the games of Frostbite and Double Dunk, but struggling on Beam Rider. Bolded font indicates statistical significance (using a student's t-test) between DRQN and our reimplementation of DQN.

3.9 MDP to POMDP Generalization

Figure 3.7 shows that DRQN performance increases when trained on a POMDP and then evaluated on a MDP. Arguably the more interesting question is the re-

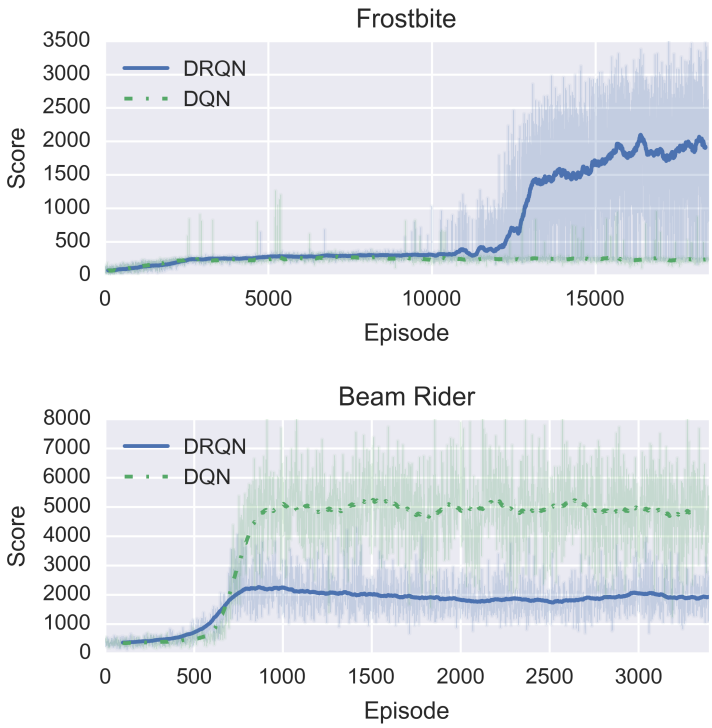


Figure 3.9: Frostbite and Beam Rider represent the best and worst games for DRQN. Frostbite performance jumps as the agent learns to reliably complete the first level.

verse: can a recurrent network be trained on a standard MDP and then generalize to a POMDP at evaluation time? To address this question, we evaluate the highest-scoring policies of DRQN and DQN over the flickering equivalents of all 9 games in Table 3.1. Figure 3.10 shows that while both algorithms incur significant performance decreases on account of the missing information, DRQN captures more of its previous performance than DQN across all levels of flickering. We conclude that recurrent controllers have a certain degree of robustness against missing information, even trained with full state information.

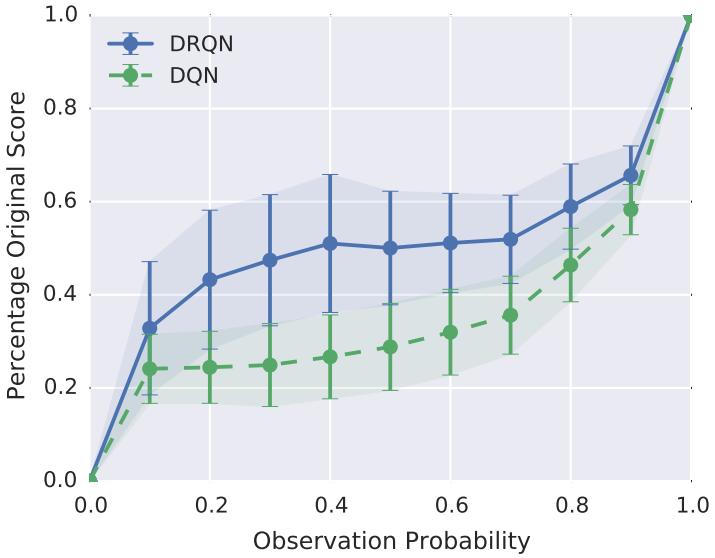


Figure 3.10: When trained on normal games (MDPs) and then evaluated on flickering games (POMDPs), DRQN's performance degrades more gracefully than DQN's. Each data point shows the average percentage of the original game score over all 9 games in Table 3.1.

3.10 Alternative Architectures

DRQN's architecture, in which the LSTM layer is located directly after the last convolutional layer (e.g. replacing DQN's IP1 layer) was the best performing architecture of a number of candidates. Alternative architectures were evalu-

ated on the game of Beam Rider. We explored the possibility of either replacing the first non-convolutional fully connected layer with an LSTM layer (LSTM replaces IP1, the DRQN architecture described in this chapter) or adding the LSTM layer between the first and second fully connected layers (LSTM over IP1). Results strongly indicated LSTM should replace IP1. We hypothesize that having LSTM replace IP1 allows LSTM direct access to the convolutional features. Additionally, adding a Rectifier layer after the LSTM layer consistently reduced performance.

Description	Percent Improvement
LSTM replaces IP1	709%
ReLU-LSTM replaces IP1	533%
LSTM over IP1	418%
ReLU-LSTM over IP1	0%

Table 3.2: Percentage improvement over ReLU-LSTM after IP1 Layer. Consistent improvements are observed when not applying ReLU activation function to LSTM outputs and also when locating the LSTM layer directly after the last convolutional layer. Row 1 corresponds to the architecture used throughout the rest of the chapter.

Another possible architecture combines frame stacking from DQN with the recurrency of LSTM. This architecture accepts a stack of the four latest frames at every timestep. The LSTM portion of the architecture remains the same and is unrolled over the last 10 timesteps. In theory, this modification should allow velocity detection to happen in the convolutional layers of the network, leaving the LSTM free to perform higher-order processing. This architecture has the largest number of parameters and requires the most training time. Unfortunately, results show that the additional parameters do not lead to increased performance on the set of games examined. It is possible that the network has too many parameters and is prone to overfitting the training experiences it has seen.

3.11 Computational Efficiency

Computational efficiency of RNNs is an important concern. We conducted experiments by performing 1000 backwards and forwards passes and reporting the

average time in milliseconds required for each pass. Experiments used a single Nvidia GTX Titan Black using CuDNN and a fully optimized version of Caffe. Results indicate that computation scales sub-linearly in both the number of frames stacked in the input layer and the number of iterations unrolled. Even so, models trained on a large number of stacked frames and unrolled for many iterations are often computationally intractable. For example a model unrolled for 30 iterations with 10 stacked frames would require over 56 days to reach 10 million iterations. Thus, the ability of an RNN to process information through time is counterbalanced by the additional computational cost when training. In practice, the choice between a recurrent and non-recurrent model will often depend on the characteristics of the domain (e.g. partial observability) and the availability of computational resources.

	Backwards (ms)			Forwards (ms)		
Frames	1	4	10	1	4	10
Baseline	8.82	13.6	26.7	2.0	4.0	9.0
Unroll 1	18.2	22.3	33.7	2.4	4.4	9.4
Unroll 10	77.3	111.3	180.5	2.5	4.4	8.3
Unroll 30	204.5	263.4	491.1	2.5	3.8	9.4

Table 3.3: Average milliseconds per backwards/forwards pass. Frames refers to the number of channels in the input image. Baseline is a non recurrent network (e.g. DQN). Unroll refers to an LSTM network backpropagated through time $1/10/30$ steps.

3.12 Related Work

Previously, LSTM networks have been demonstrated to solve POMDPs when trained using policy gradient methods (Wierstra et al., 2007). In contrast to policy gradient, our work uses temporal-difference updates to bootstrap an action-value function. Additionally, by jointly training convolutional and LSTM layers we are able to learn directly from pixels and do not require hand-engineered features.

LSTM has been used as an advantage-function approximator and shown to solve a partially observable corridor and cartpole tasks better better than comparable

(non-LSTM) RNNs (Bakker, 2001). While similar in principle, the corridor and cartpole tasks feature tiny states spaces with just a few features.

In parallel to our work, (Narasimhan et al., 2015) independently combined LSTM with Deep Reinforcement Learning to demonstrate that recurrency helps to better play text-based fantasy games. The approach is similar but the domains differ: despite the apparent complexity of the fantasy-generated text, the underlying MDPs feature relatively low-dimensional manifolds of underlying state space. The more complex of the two games features only 56 underlying states. Atari games, in contrast, feature a much richer state space with typical games having millions of different states. However, the action space of the text games is much larger with a branching factor of 222 versus Atari's 18.

3.13 Chapter Summary

Real-world tasks often feature incomplete and noisy state information, resulting from partial observability. This chapter described our work to modify DQN to handle the noisy observations characteristic of POMDPs by combining a Long Short Term Memory with a Deep Q-Network. The resulting *Deep Recurrent Q-Network* (DRQN), despite seeing only a single frame at each step, is still capable of integrating information across frames to detect relevant information such as velocity of on-screen objects. Additionally, on the game of Pong, DRQN is better equipped than a standard Deep Q-Network to handle the type of partial observability induced by flickering game screens.

Furthermore, when trained with partial observations, DRQN can generalize its policies to the case of complete observations. On the Flickering Pong domain, performance scales with the observability of the domain, reaching near-perfect levels when every game screen is observed. This result indicates that the recurrent network learns policies that are robust enough to handle missing game screens and scalable enough to improve performance as observability increases. Generalization also occurs in the opposite direction: when trained on standard Atari games and evaluated against flickering games, DRQN's performance generalizes better than

DQN's at all levels of partial information.

Our experiments suggest that Pong represents an outlier among the examined games. Across a set of ten Flickering MDPs we observe no systematic improvement when employing recurrency. Similarly, across non-flickering Atari games, there are few significant differences between the recurrent and non-recurrent player. This observation leads us to conclude that while recurrency is a viable method for handling state observations, it confers no systematic benefit compared to stacking the observations in the input layer of a convolutional network. One avenue for future research is identifying the relevant characteristics of Pong and Frostbite that lead to better performance by recurrent networks.

Chapter 4
Deep RL in Parameterized Action Space

Building on the fundamentals presented in Chapter 2 as well as the DDPG algorithm (Chapter 2.7), this chapter describes a successful application of deep reinforcement learning to the Half Field Offense Domain. This chapter assumes familiarity with the DDPG algorithm presented in Chapter 2.7 as well as the Half Field Offense domain (Chapter 2.5). This chapter contributes to the portion of the thesis question concerned with leveraging deep neural networks towards domains that feature continuous parameterized action space and forms the basis of thesis contribution 3: Deep RL in parameterized action space.[1]

Specifically, this chapter extends the Deep Deterministic Policy Gradients (DDPG) algorithm (Lillicrap et al., 2015) (discussed in Chapter 2.7) into a parameterized action space. We document two modifications to the published version of the DDPG algorithm: namely bounding action space gradients and on-policy updates. We found these modifications necessary for stable learning in this domain and we expect they will be valuable for future practitioners attempting to learn in continuous, bounded action spaces.

Furthermore, this chapter demonstrates reliable learning, from scratch, of RoboCup soccer policies capable of goal scoring. These policies operate on a low-level continuous state space and a parameterized-continuous action space. Using a single reward function, the agents learn to locate and approach the ball, dribble to the goal, and score on an empty goal. The best learned agent proves more reliable at scoring goals, though slower, than the hand-coded 2012 RoboCup champion.

Half-Field-Offense (see Chapter 2.5 for more details) is a research platform for exploring single agent learning, multi-agent learning, and adhoc teamwork. HFO features a low-level continuous state space and parameterized-continuous action space. Specifically, the parameterized action space requires the agent to first

[1]This chapter is based in part on the following publication (Hausknecht and Stone, 2016a). I am the primary author of the research in this chapter and the related publication.

select the type of action it wishes to perform from a discrete list of high level actions and then specify the continuous parameters to accompany that action. This parameterization introduces structure not found in a purely continuous action space.

The rest of this chapter is organized as follows: we first present the reward signal used in the HFO empty goal task. Next we discuss the architecture used for the parameterized action space learning. Section 4.5 presents a method of bounding action space gradients, and Section 4.8 describes a method for mixing off-policy and on-policy updates. Finally, Section 4.7 covers experiments and results.

4.1 Reward Signal

True rewards in the HFO domain come from winning full games. However, such a reward signal is far too sparse for learning agents to gain traction. Instead we introduce a hand-crafted reward signal with four components: **Move To Ball Reward** provides a scalar reward proportional to the change in distance between the agent and the ball $d(a, b)$. An additional reward $\mathbb{I}^{kick}$ of 1 is given the first time the agent is close enough to kick the ball. **Kick To Goal Reward** is proportional to the change in distance between the ball and the center of the goal $d(b, g)$. An additional reward is given for scoring a goal $\mathbb{I}^{goal}$. A weighted sum of these components results in a single reward that first guides the agent close enough to kick the ball, then rewards for kicking towards goal, and finally for scoring. It was necessary to provide a higher gain for the kick-to-goal component of the reward because immediately following each kick, the move-to-ball component produces negative rewards as the ball moves away from the agent. The overall reward is as follows:

$$r_t = d_{t-1}(a, b) - d_t(a, b) + \mathbb{I}_t^{kick} + 3\big(d_{t-1}(b, g) - d_t(b, g)\big) + 5\mathbb{I}_t^{goal} \tag{4.1}$$

In addition to potentially biasing the optimal policy, it is disappointing that reward engineering is necessary. However, the exploration task proves far too difficult to ever gain traction on a reward that consists only of scoring goals, because

acting randomly is exceedingly unlikely to yield even a single goal in any reasonable amount of time. An interesting direction for future work is to find better ways of exploring large state spaces. One recent approach in this direction, (Stadie et al., 2015) assigned exploration bonuses based on a model of system dynamics.

4.2 Network Architecture

The architecture and update for DDPG is discussed in Chapter 2.7 and depicted in Figure 2.3. Both the actor and critic employ the same architecture: The 58 state inputs are processed by four fully connected layers consisting of 1024-512-256-128 units respectively. Each fully connected layer is followed by a rectified linear (ReLU) activation function with negative slope 10^{-2}. Weights of the fully connected layers use Gaussian initialization with a standard deviation of 10^{-2}. Connected to the final inner product layer are two linear output layers: one for the four discrete actions and another for the six parameters accompanying these actions. In addition to the 58 state features, the critic also takes as input the four discrete actions and six action parameters. It outputs a single scalar Q-value. We use the ADAM solver with both actor and critic learning rate set to 10^{-3}. Target networks track the actor and critic using a $\tau = 10^{-4}$. Complete source code for our agent is available at `https://github.com/mhauskn/dqn-hfo` and for the HFO domain at `https://github.com/mhauskn/HFO/`. Building on this architecture, we now present the parameterized action space.

4.3 Parameterized Action Space Architecture

Following notation in (Masson and Konidaris, 2015), a Parameterized Action Space Markov Decision Process (PAMDP) is defined by a set of discrete actions $A_d = \{a_1, a_2, \ldots, a_k\}$. Each discrete action $a \in A_d$ features m_a continuous parameters $\{p^a_1, \ldots, p^a_{m_a}\} \in \mathbb{R}^{m_a}$. Actions are represented by tuples $(a, p^a_1, \ldots, p^a_{m_a})$. Thus the overall action space $A = \cup_{a \in A_d}(a, p^a_1, \ldots, p^a_{m_a})$.

In Half Field Offense, the complete parameterized action space (Section

2.5.2) is $A = (\text{Dash}, p_1^{\text{dash}}, p_2^{\text{dash}}) \cup (\text{Turn}, p_3^{\text{turn}}) \cup (\text{Tackle}, p_4^{\text{tackle}}) \cup (\text{Kick}, p_5^{\text{kick}}, p_6^{\text{kick}})$. The actor network in Figure 2.3 factors the action space into one output layer for discrete actions $(\text{Dash}, \text{Turn}, \text{Tackle}, \text{Kick})$ and another for all six continuous parameters $(p_1^{\text{dash}}, p_2^{\text{dash}}, p_3^{\text{turn}}, p_4^{\text{tackle}}, p_5^{\text{kick}}, p_6^{\text{kick}})$.

4.4 Action Selection and Exploration

Using the factored action space, deterministic action selection proceeds as follows: At each timestep, the actor network outputs values for each of the four discrete actions as well as six continuous parameters. The discrete action is chosen to be the maximally valued output $a = \max(\text{Dash}, \text{Turn}, \text{Tackle}, \text{Kick})$ and paired with associated parameters from the parameter output layer $(a, p_1^a, \ldots, p_{m_a}^a)$. Thus the actor network simultaneously chooses which discrete action to execute and how to parameterize that action.

During training, the critic network receives, as input, the values of the output nodes of all four discrete actions and all six action parameters. We do not indicate to the critic which discrete action was actually applied in the HFO environment or which continuous parameters are associated with that discrete action. Similarly, when updating the actor, the critic provides gradients for all four discrete actions and all six continuous parameters. While it may seem that the critic is lacking crucial information about the structure of the action space, our experimental results in Section 4.7 demonstrate that the critic learns to provide gradients to the correct parameters of each discrete action.

Exploration in continuous action space differs from that in discrete space. We adapt ϵ-greedy exploration to parameterized action space: with probability ϵ, a random discrete action $a \in A_d$ is selected and the associated continuous parameters $\{p_1^a, \ldots, p_{m_a}^a\}$ are sampled using a uniform random distribution. Experimentally, we anneal ϵ from 1.0 to 0.1 over the first $10,000$ updates. (Lillicrap et al., 2015) demonstrate that Ornstein-Uhlenbeck exploration is also successful in continuous action space.

4.5 Bounded Parameter Space Learning

The Half Field Offense domain bounds the range of each continuous parameter. Parameters indicating direction (e.g. Turn and Kick direction) are bounded in $[-180, 180]$ and parameters for power (e.g. Kick and Dash power) are bounded in $[0, 100]$. Without enforcing these bounds, after a few hundred updates, we observed continuous parameters routinely exceeding the bounds. If updates were permitted to continue, parameters would quickly trend towards astronomically large values. This problem stems from the critic providing gradients that encourage the actor network to continue increasing a parameter that already exceeds bounds. We explore three approaches for preserving parameters in their intended ranges:

Zeroing Gradients: Perhaps the simplest approach is to examine the critic's gradients for each parameter and zero the gradients that suggest increasing/decreasing the value of a parameter that is already at the upper/lower limit of its range:

$$\nabla_p = \begin{cases} \nabla_p & \text{if } p_{\min} < p < p_{\max} \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

Where ∇_p indicates the critic's gradient with respect to parameter p, (e.g. $\nabla_p Q(s_t, a|\theta^Q)$) and $p_{\min}, p_{\max}, p$ indicate respectively the minimum bound, maximum bound, and current activation of that parameter.

Squashing Function: The hyperbolic tangent (tanh) squashing function is used to bound the activation of each parameter. Subsequently, the parameters are rescaled into their intended ranges. This approach has the advantage of not requiring manual gradient tinkering ($\nabla_p = \nabla_p$), but presents issues if the squashing function saturates.

Inverting Gradients: This approach captures the best aspects of the zeroing and squashing gradients, while minimizing the drawbacks. Gradients are downscaled as the parameter approaches the boundaries of its range and are inverted if the parameter exceeds the value range. This approach actively keeps parameters within bounds while avoiding problems of saturation. For example, if the critic continually recommends increasing a parameter, it will converge to the parameter's upper

bound. If the critic then decides to decrease that parameter, it will decrease immediately. In contrast, a squashing function would be saturated at the upper bound of the range and require many updates to decrease. Mathematically, the inverted gradient approach may be expressed as follows:

$$\nabla_p = \nabla_p \cdot \begin{cases} (p_{\max} - p)/(p_{\max} - p_{\min}) & \text{if } \nabla_p \text{ suggests increasing } p \\ (p - p_{\min})/(p_{\max} - p_{\min}) & \text{otherwise} \end{cases} \tag{4.3}$$

It should be noted that these approaches are not specific to HFO or parameterized action space. Any domain featuring a bounded-continuous action space will require a similar approach for enforcing bounds. All three approaches are empirically evaluated the next section.

4.6 Gradient Bounding Results

We evaluate the zeroing, squashing, and inverting gradient approaches in the parameterized HFO domain on the task of approaching the ball and scoring a goal. For each approach, we independently train two agents. All agents are trained for 3 million iterations, approximately 20,000 episodes of play. Training each agent took three days on a NVidia Titan-X GPU.

Of the three approaches, only the inverting gradient shows robust learning. Indeed both inverting gradient agents learned to reliably approach the ball and score goals. None of the other four agents using the squashing or zeroing gradients were able to reliably approach the ball or score.

Further analysis of the squashing gradient approach reveals that parameters stayed within their bounds, but squashing functions quickly became saturated. The resulting agents take the same discrete action with the same maximum/minimum parameters each timestep. Given the observed proclivity of the critic's gradients to push parameters towards ever larger/small values, it is no surprise that squashing functions quickly become saturated and never recover.

Further analysis of the zeroing gradient approach reveals two problems: 1) parameters still overflow their bounds and 2) instability: While the gradient zeroing approach negates any direct attempts to increase a parameter p beyond its bounds, we hypothesize the first problem stems from gradients applied to other parameters $p_i \neq p$ which inadvertently allow parameter p to overflow. Empirically, we observed learned networks attempting to dash with a power of 120, more than the maximum of 100. It is reasonable for a critic network to encourage the actor to dash faster.

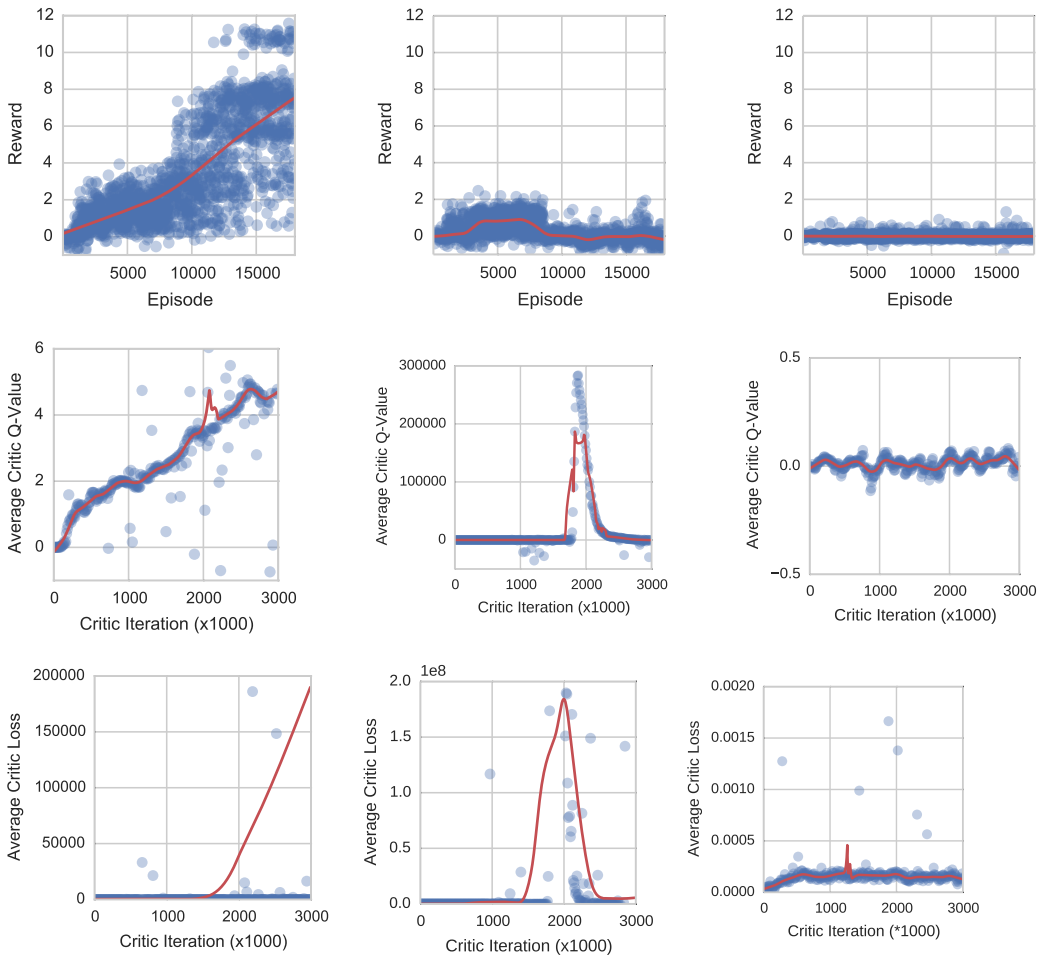
Unstable learning was observed in one of the two zeroing gradient agents. This instability is well captured in the Q-Values and critic losses shown in Figure 4.1. It's not clear why this agent became unstable, but the remaining stable agent showed clear results of not learning.

These results highlight the necessity of non-saturating functions that effectively enforce action bounds. The approach of inverting gradients was observed to respect parameter boundaries (observed dash power reaches 98.8 out of 100) without saturating. As a result, the critic was able to effectively shape the actor's policy. Further evaluation of the reliability and quality of the inverting-gradient policies is presented in the next section.

4.7 Single Agent Learning

We further evaluate the inverting gradient agents by comparing them to an expert agent independently created by the Helios RoboCup-2D team. This agent won the 2012 RoboCup-2D world championship and source code was subsequently released (Akiyama, 2010). Thus, this hand-coded policy represents an extremely competent player and a high performance bar.

As an additional baseline we compare to a SARSA learning agent. State-Action-Reward-State-Action (SARSA) is an algorithm for model-free on-policy Reinforcement Learning (Sutton and Barto, 1998). The SARSA agent learns in a simplified version of HFO featuring high-level discrete actions for moving, dribbling, and shooting the ball. As input, it is given continuous features that include the distance and angle to the goal center. Tile coding (Sutton and Barto, 1998) is used



(a) Inverting Gradients (b) Zeroing Gradients (c) Squashing Gradients

Figure 4.1: **Analysis of gradient bounding strategies:** The left/middle/right columns respectively correspond to the inverting/zeroing/squashing gradients approaches to handling bounded continuous actions. **First row** depicts overall task performance: Only the inverting gradient approach succeeds in learning the soccer task. **Second row** shows average Q-Values produced by the critic throughout the entire learning process: Inverting gradient approach shows smoothly increasing Q-Values. Zeroing approach shows instability in the critic. The squashing approach shows stable Q-Values that accurately reflect the actor's performance. **Third row** shows the average loss experienced during a critic update (Equation 2.9): As more reward is experienced critic loss is expected to rise as past actions are seen as increasingly sub-optimal. Inverting gradients shows growing critic loss. Zeroing gradients approach shows unstably large loss. Squashing gradients never discovers much reward and loss stays near zero.

to generate state features over the continuous state space. Experiences collected by playing the game are then used to bootstrap a value function.

To show that the deep reinforcement learning process is reliable, in addition to the previous two inverting-gradient agents we independently train another five inverting-gradient agents, for a total of seven agents DDPG_{1-7}. All seven agents learned to score goals. Comparing against the Helios' champion agent, each of the learned agents is evaluated for 100 episodes on how quickly and reliably it can score.

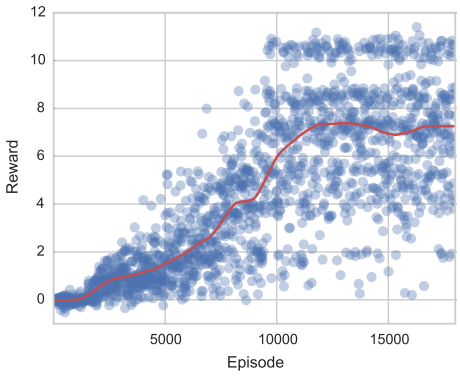
Six of seven DDPG agents outperform the SARSA baseline, and remarkably, three of the seven DDPG agents score more reliably than Helios' champion agent. Occasional failures of the Helios agent result from noise in the action space, which occasionally causes missed kicks. In contrast, DDPG agents learn to take extra time to score each goal, and become more accurate as a result. This extra time is reasonable considering DDPG is rewarded only for scoring and experiences no real pressure to score more quickly. We are encouraged to see that deep reinforcement learning can produce agents competitive with and even exceeding an expert handcoded agent.

4.8 Mixing On-Policy and Off-Policy Updates

Temporal-Difference (TD) methods learn online directly from experience, do not require a model of the environment, offer guarantees of convergence to optimal performance, and are straightforward to implement (Sutton and Barto, 1998). For all of these reasons, TD learning methods have been widely used since the inception of reinforcement learning. Like TD methods, Monte Carlo methods also learn online directly from experience. However, unlike TD-methods, Monte Carlo methods do not bootstrap value estimates and instead learn directly from returns. Figure 4.3 shows the relationship between these methods.[2]

In this section, we focus on two methods: on-policy Monte Carlo (Sutton and Barto, 1998) and Q-Learning (Watkins and Dayan, 1992). On-policy MC employs

[2]This section is based in part on the work published at (Hausknecht and Stone, 2016b).



(a) Learning Curve

	Scoring Percent	Avg. Steps to Goal
Helios' Champion	.96	72.0
SARSA	.81	70.7
$DDPG_1$	1	108.0
$DDPG_2$	.99	107.1
$DDPG_3$	.98	104.8
$DDPG_4$	.96	112.3
$DDPG_5$	.94	119.1
$DDPG_6$	.84	113.2
$DDPG_7$	.80	118.2

(b) Evaluation Performance

Figure 4.2: **Left:** Scatter plot of learning curves of DDPG-agents with Lowess curve. Three distinct phases of learning may be seen: the agents first get small rewards for approaching the ball (episode 1500), then learn to kick the ball towards the goal (episodes 2,000 - 8,000), and start scoring goals around episode 10,000. **Right:** DDPG-agents score nearly as reliably as expert baseline, but take longer to do so. A video of $DDPG_1$'s policy may be viewed at `https://youtu.be/Ln0Cl-jE_40`.

on-policy updates without any bootstrapping, while Q-Learning uses off-policy updates with bootstrapping. Both algorithms seek to estimate the action-value function $Q(s, a)$ directly from experience tuples of the form $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ and both provably converge to optimality so long as all state-value pairs are visited an infinite number of times and the behavior policy eventually becomes greedy. Both methods are driven by temporal difference updates which take the following form, where y is the update target and α is a stepsize:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(y - Q(s_t, a_t))$$

The main difference between these methods may be understood by examining their update targets. The update targets for Q-Learning, n-step-Q-learning, and on-policy MC may be expressed as follows:

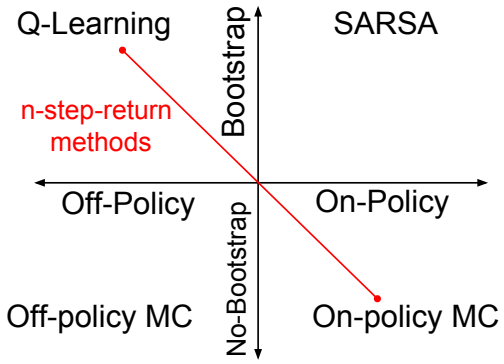


Figure 4.3: Relationship of RL algorithms according to whether they bootstrap the value function and if they are on or off-policy. This work compares Q-Learning updates with On-Policy Monte-Carlo updates. N-step-reward methods such as N-step-Q-Learning bridge the spectrum between Q-Learning and on-policy Monte-Carlo.

$$y_{\text{q-learning}} = r_t + \gamma \max_a Q(s_{t+1}, a)$$
$$y_{\text{n-step-q}} = r_t + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n \max_a Q(s_{t+n+1}, a)$$
$$y_{\text{on-policy-monte-carlo}} = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$$

As seen in the update, the on-policy MC targets are estimated directly from the rewards received in the experience tuples. In contrast the Q-Learning target truncates the reward sequence with its own value estimate.

One way of relating Q-Learning to on-policy MC is to consider n-step-return methods. These methods make use of multi-step returns and are potentially more efficient at propagating rewards to relevant state-action pairs. On-policy MC is realized when n approaches infinity (or maximum episode length). Recently, multi-step returns have been shown to be useful in the context of deep reinforcement learning (Mnih et al., 2016).

4.8.1 Motivation for On-Policy Updates

Using deep neural networks to approximate the value function is a double-edged sword. Deep networks are powerful function approximators and strongly generalize between similar state inputs. However, generalization can cause divergence in the case of repeated boostrapped temporal-difference updates. Let us consider the case of the same Q-Learning update applied to a deep neural network parameterized by θ:

$$Q(s_t, a_t|\theta) = r_{t+1} + \gamma \max_a Q(s_{t+1}, a|\theta)$$

If it is the case that $r_{t+1} > 0$ and s_t is similar to s_{t+1}, then Q-Value estimates will quickly diverge to infinity as this update is repeated. The divergence is because the network's generalization causes the estimate of s_{t+1} to grow with the estimate of s_t, causing update targets to continually grow.

To address this problem a target network is used to make bootstrap updates more stable (Mnih et al., 2015).[3] By updating the target network at a slower rate than the main network, it is possible to limit the generalization from s_{t+1} to s_t, stabilize the update targets, and prevent Q-Value divergence. Using τ (typically $.001$) to govern the rate that the target network follows the main network, the same update using a target network $\hat{\theta}$ takes the following form:

$$Q(s_t, a_t|\theta) = r_{t+1} + \gamma \max_a Q(s_{t+1}, a|\hat{\theta})$$

$$\hat{\theta} = \tau\theta + (1 - \tau)\hat{\theta}$$

On-policy Monte Carlo updates remove the need for a target network since the target is computed directly from the rewards of the trajectory rather than bootstrapped. Such an update makes sense particularly when there is reason to believe that the neural network's estimates of the next state $Q(s_{t+1}, a|\hat{\theta})$ are inaccurate, as is typically the case when learning begins. Additionally, Monte Carlo update targets cannot diverge since they are bounded by the actual rewards received. However, on-

[3] Another way to address this problem is to not repeatedly update the same experience tuple.

policy MC updates suffer from the problem that exploratory actions may negatively skew Q-Value estimates. We now address the issue of how to efficiently compute on-policy MC targets.

4.8.2 Computing On-Policy MC Targets

We store on-policy targets y_t in the replay memory by augmenting each transition to include the on-policy target: $(s_t, a_t, r_t, y_t, s_{t+1}, a_{t+1})$. As shown in Algorithm 2, we first accumulate a full episode of experience tuples then work backward to compute on-policy targets and add augmented experiences to the replay memory. Once stored in the replay memory, on-policy MC targets can be accessed directly from the augmented experience tuples without requiring any additional computation.

Algorithm 2 Compute On-Policy MC Targets

Given: Trajectory $T_{0\ldots n}$, Replay Memory $\mathcal{D}$
$R \leftarrow 0$
for $t \in \{n \ldots 0\}$ **do**
    $R \leftarrow r_t + \gamma R$
    $y_t \leftarrow R$
    $\mathcal{D} \leftarrow (s_t, a_t, r_t, y_t, s_{t+1}, a_{t+1})$

4.8.3 Mixing Update Targets

Rather than using exclusively on-policy or off-policy targets it is possible, and in many cases desirable, to mix on-policy MC targets with off-policy 1-step Q-Learning targets. Mixing is accomplished using a β parameter in $[0, 1]$. The overall mixed update target is expressed as follows:

$$y = \beta\, y_{\text{on-policy-MC}} + (1 - \beta)\, y_{\text{q-learning}}$$

Like n-step-return methods, mixed targets present a way to tradeoff between on-policy MC updates, and off-policy bootstrap updates. This way of mixing targets

could be considered a special case of eligibility traces (Sutton and Barto, 1998): eligibility traces mix all n-step returns while this method mixes the one-step return with the Monte Carlo return. In principle, mixing all n-step returns is more general, but requires much more computation to execute in practice. In particular, each n-step return must be truncated by a Q-Value estimate, which requires a forward pass through the Critic network. Thus, computing all n-step returns for an episode requires n forward passes from the critic network. While it would be possible to reuse these n-step returns if sequentially updating every experience in an episode, in practice it is beneficial to randomly sample experiences from a replay memory that spans multiple episodes (see Chapter 2.6 for more details). This random sampling makes it difficult to perform an efficient eligibility trace update.

Therefore we choose to sacrifice the precision gained from mixing all n-step returns in favor of the efficiency from just mixing the one-step and Monte-Carlo targets. This approach requires only a single forward pass over the critic network to estimate the Q-Value of the 1-step target and empirically yields good results.

The next sections present results using mixed update targets for the cases of discrete action space learning using DQN and continuous action space learning using DDPG.

4.8.4 Scoring on a Goalie

As in real soccer, the task of scoring on a goal keeper is far more difficult that that of scoring on an empty goal. The goal keeper's policy was independently programmed by Helios RoboCup 2D team (Akiyama, 2010) and is highly adept. The keeper continually re-positions itself to prevent easy shots and charges the striker if it nears the goal. The keeper blocks any shots within its reach, but the size of the goal allows a correctly positioned striker to score with a precise kick.

We modify the task initialization in order to emphasize goal scoring rather than approaching the ball. Specifically, to begin each episode we initialize the agent three fifths of the way down the field and give it possession of the ball. The agent must learn to dribble and position itself as well as learn to precisely kick the ball at open goal angles. Rewards in this task are the same as in the empty goal task: the

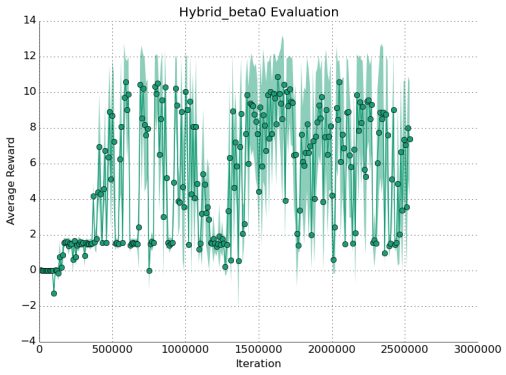
agent is rewarded for approaching the ball, moving the ball towards the goal, and scoring.

Results in Figure 4.5 show that mixed updates are not only more stable and higher performing, they are also necessary to learn to reliably score on the keeper. Preferring off-policy targets yields the best performance on this task with $\beta = 0.2$ exhibiting the fastest learning and and a final policy that successfully scores goals every time in an evaluation consisting of 100 episodes. In contrast, the expert hand-coded Helios offense agent scores 81.4% of the time against the keeper. This offense agent was programmed by an independent team of human experts specifically for the task of RoboCup 2D soccer. That it is significantly outperformed by a learned agent is a testament to power of modern deep reinforcement learning methods. A video of the learned policy may be viewed at `https://youtu.be/JEGMKvAoB34`.

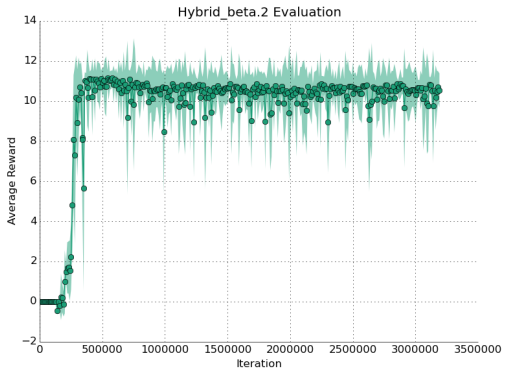
4.9 Chapter Summary

This chapter presented two extensions of the DDPG algorithm that allowed it to better handle the parameterized action space of the Half Field Offense domain. Inverting action gradients, the first extension was motivated by the bounded nature of the continuous action space in HFO. In order to encourage DDPG to respect the bounds on each continuous HFO action, we presented an approach for inverting action gradients. This approach inspects the gradients that flow from the critic to the actor and modifies them in such a way that the actions smoothly approach the upper or lower bounds of their range and without saturating. Compared to two alternative approaches - squashing or zeroing gradients, the inverting gradients shows the quickest and most stable learning on the HFO task of scoring on an empty goal.

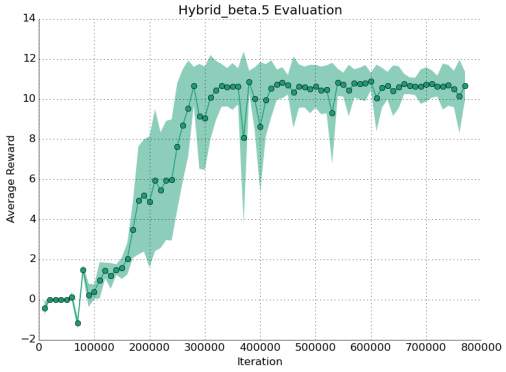
The second extension two DDPG was mixing on-policy updates with off-policy targets. Standard DDPG uses an approximate off-policy update in which the update target is computed by an approximate max over next state actions. We demonstrated a computationally efficient method for computing on-policy Monte



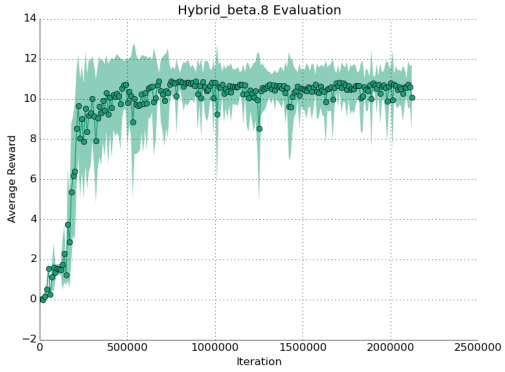
(a) $\beta = 0$



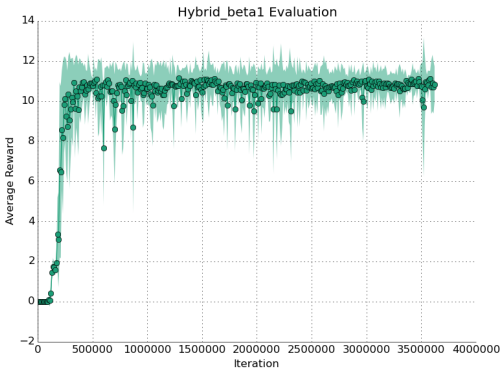
(b) $\beta = 0.2$



(c) $\beta = 0.5$

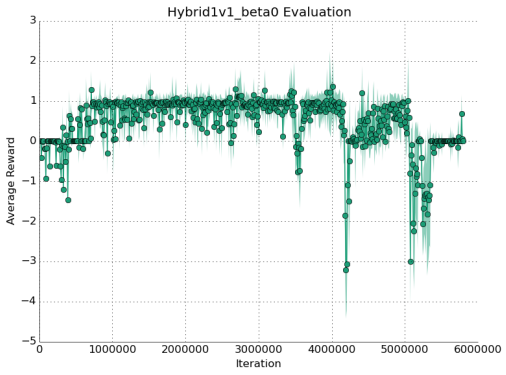


(d) $\beta = 0.8$

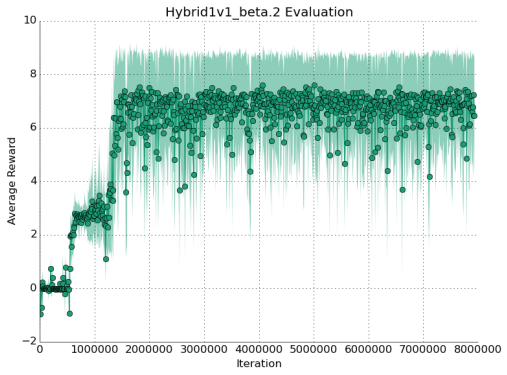


(e) $\beta = 1$

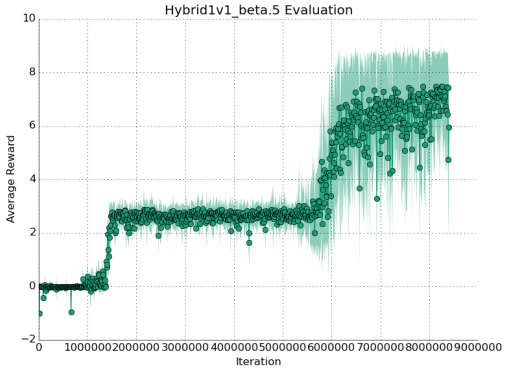
Figure 4.4: **Performance of mixed-updates on empty goal task**: The maximum possible reward is 11. Purely off-policy updates ($\beta = 0$) achieve this maximum reward but show inconsistent performance. All of the mixed updates achieve the maximum task performance with pure Monte Carlo ($\beta = 1$) doing best. Note that the scale of the y-axis changes between plots.



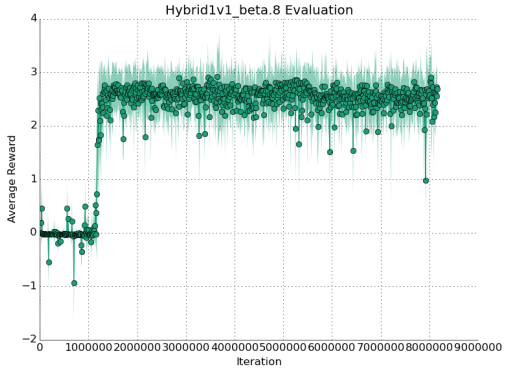
(a) $\beta = 0$



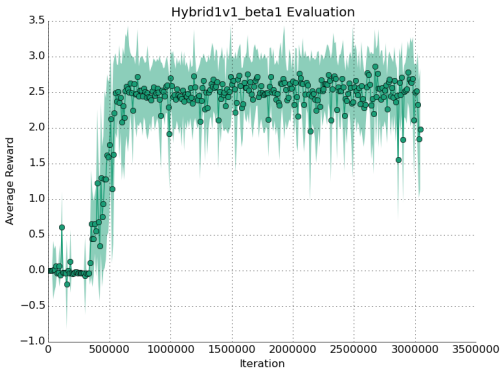
(b) $\beta = 0.2$



(c) $\beta = 0.5$



(d) $\beta = 0.8$



(e) $\beta = 1$

Figure 4.5: **Performance of mixed-updates on Keeper task**: On this task, only mixed-updates ($\beta = 0.2, 0.5$) achieve the maximum reward of 6 and are able to learn to reliably score on the keeper. Off-policy updates ($\beta = 0$) and on-policy updates ($\beta = 0.8, 1$) never reliably learn to score. Note that the scale of the y-axis changes between plots.

Carlo update targets and show that by mixing the standard off-policy targets with the on-policy targets, superior performance and stability is achieved on the empty goal HFO task as well as a 1v1 HFO task. In fact, only by including the on-policy Monte Carlo targets was the agent able to learn to reliably score on a goal keeper. These two extensions are used throughout the future chapters of this thesis.

Chapter 5
Multiagent Deep Reinforcement Learning

In order to get the most from this chapter, a working understanding Chapter 4 is beneficial. However, it is possible to interpret the multiagent architectures without a full understanding of DDPG or parameterized action space learning. This chapter addresses the portion of the thesis question regarding how multiple deep reinforcement learning agents can learn to cooperate in a multiagent setting. It forms the basis for thesis contribution 4: An exploration of multiagent deep RL.

Multiagent reinforcement learning (Littman, 1994; Tan, 1998) has taken shape as an active area of research in its own right. Other topics of interest in a multiagent environment, such as coordination (Tambe, 1997) and ad hoc teamwork (Stone et al., 2010) have also been actively pursued. This chapter presents several approaches for encouraging cooperation and coordination between multiple deep reinforcement learning agents in the Half Field Offense domain. Fortunately, soccer is a cooperative task because the offense and defense team benefit from using strategies that involve passing the ball to other players and teammate-aware positioning. However, cooperative behaviors are not easily learned by a single agent and, for human soccer players, require practice before they may be executed smoothly. This chapter first presents a multiagent version of the empty goal task and then presents approaches designed to facilitate cooperation between the learning agents.

5.1 Multiagent Empty Goal Task

We introduce a multiagent version of the empty goal HFO task whose reward function is given in Chapter 4.1. Similar to the single agent version, multiagent empty goal requires the agents to locate and approach the ball, dribble the ball to the goal, and score a goal. However, the multiagent version of the task modifies the reward signal in three significant ways: first, the lion's share of the goal scoring reward (5 points) is given to the agent who kicked the ball into the goal. One

point of reward is provided to the teammate. Second, only the agent who currently possesses the ball is rewarded for kicking the ball towards the goal. Third, as long as neither agent has control of the ball, both agents are rewarded for moving towards the ball. However, as soon as one agent has control of the ball, only it is rewarded for moving towards the ball. Control of the ball is gained by being close enough to kick the ball. This asymmetric reward function provides greater rewards to an agent that is actively participating in the scoring process, while still providing small rewards to the teammate. The reward function may be expressed as follows:

$$r_t = \begin{cases} d_{t-1}(a,b) - d_t(a,b) + \mathbb{I}_t^{kick} + 3\big(d_{t-1}(b,g) - d_t(b,g)\big) + 5\mathbb{I}_t^{goal} & \text{if on-ball} \\ d_{t-1}(a,b) - d_t(a,b) & \text{no agent on-ball} \\ \mathbb{I}^{goal} & \text{teammate on-ball} \end{cases} \tag{5.1}$$

In this multiagent task, we recognize several different measures of success. At a minimum we desire at least one of the agents to learn a robust scoring behavior. However, we prefer if both agents learn to become competent at the task. The highest achievement would be if the agents can learn to actively share with each other by passing the ball and speeding up the scoring process.

5.2 Cooperative vs. Non-Cooperative Tasks

The distinction between cooperative and non-cooperative tasks is an important one. For a task to be cooperative, it must feature a reward function that provides maximal reward to each agent when that agent is acting cooperatively. In other words, neither agent can maximize rewards by acting non-cooperatively.

Because of the one point of goal reward given to the teammate, both agents are encouraged to find policies that reliably result in goals being scored. Thus, this task may be considered cooperative. However, analyzing the multiagent soccer task from a game theoretic perspective, each agent is most highly rewarded if it is the only competent member of the team. Indeed, as shown in Chapter 4, this task can

be fully solved (in the sense that a goal is scored every episode) by a single agent. In this light, there is little incentive for a competent agent to share reward with a teammate, since it stands to lose reward.

Thus, this task is cooperative in the sense that both agents are incentivized to make sure goals are reliably scored, but not so cooperative that the agents are rewarded for passing. Indeed agents receive maximum reward for monopolizing control of the ball. Having set the context for this task, the next sections present several approaches for solving cooperative multiagent tasks.

5.3 Independent Learning Baseline

In the independent learning baseline, agents learn at the same time in the same task, but employ no sophisticated techniques to encourage coordination or cooperation. Whether or not they eventually cooperate depends upon the nature of the task and the difficulty of learning cooperative behavior in that task. Specifically, in the independent learning case, both agents maintain their own actor-critic networks, replay memories, and perform their own updates. The activity and behavior of the other agent is transmitted only through state features. In particular, each agent cannot directly observe the actions taken by other agents. Instead, only the effects of the action are observable (e.g. if a teammate kicks the ball, the agent observes the ball gain velocity but will have to infer that a kick action was taken). The difficulties of finding cooperative behavior are compounded by the nonstationarity of each agent's policy. Deep reinforcement learning already uses techniques such as the target network to increase learning stability in the single learning agent case. It is reasonable to expect that nonstationarity introduced by multiple learning agents could compound the difficulty of stable learning. For these reasons, the independent learning approach serves as a baseline.

5.4 Centralized Control

A second, more challenging baseline is centralized control - a single agent that operates by concatenating the state space and action spaces of all agents. The centralized control approach violates the assumption of autonomy that is normally required of each player in domains such as HFO.[1] The centralized control approach removes the nonstationarity encountered by having separate policies for each learning agent. Essentially, the centralized controller converts a multiagent task into a single agent task with and expanded state and action space. However, the centralized controller will likely encounter difficulty from learning in the expanded state space and action space. Nonetheless, the centralized controller represents a sophisticated baseline that we expect will be able to directly learn a solution that maximizes overall total reward. Figure 5.1 depicts the centralized architecture.

5.5 Parameter Sharing

Sharing parameters of the neural networks between the different agents was shown to benefit DDRQN agents who were tasked with solving riddles (Foerster et al., 2016a). Fundamentally, sharing parameters encourages similar behaviors between the agents. In the extreme case, if the agents shared all weights, they would exhibit exactly the same policy, and their actions would only differ as a function of the different states they encountered.

Depending on the domain, it may or may not be advantageous to have agents with identical policies. In the real game of soccer, players assume different roles - each of which has a different optimal policy. However, for straightforward HFO tasks such as scoring on the empty goal, cooperative behavior may be achieved by identical policies.

However, a general parameter sharing approach is not limited to share all the parameters or none of them. Instead we share the parameters for a fixed number of

[1] Of note, the small sized RoboCup league features centralized control. However, most of the standard RoboCup leagues including Standard Platform, 3D-simulation, and 2D-simulation require each player to operate autonomously.

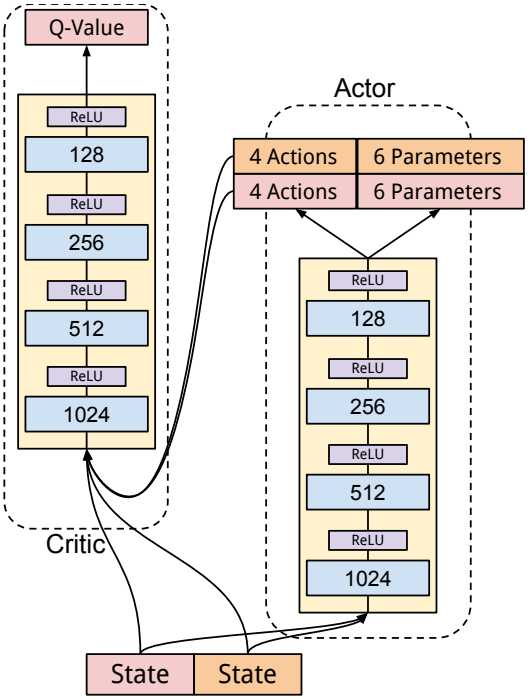


Figure 5.1: **The Centralized Architecture**: combines the state perceptions and actions spaces of two or more agents into a single decision making entity. This approach circumvents the nonstationarity of multiagent learning by lumping multiple agents into a single entity. The increased state action space size may make learning more difficult.

layers. Figure 5.2 depicts two agents sharing parameters for two layers. By sharing parameters from selected layers, the agents can strike a balance between having unique policies and sharing parameters.

In choosing which layers of the network to share parameters, we suspect that sharing parameters between the lower layers of the networks is more beneficial than sharing parameters in the upper layers of the networks. The lower layers are responsible for basic processing of state features while the upper layers are more instrumental in choosing which actions are selected by the network. To share the upper layers of the networks would make little sense as the processing in the upper layers is dependent on the parameters of the lower layers. In the best case, if both agents had the same lower layers, then the result would be identical polices. In the worst case, the parameters of the lower layers of each agent would be different and enforcing shared upper layers would result in unstable policies. To avoid this problem, we advocate sharing parameters starting with the lowest layers of the network and going upwards. Sharing parameters in this manner encourages the agents to have the same low-level processing of state features but still allows specialization in the higher layers of the network, allowing each agent to develop a unique policy.

Besides encouraging similarity between agents, sharing also has the advantage of allowing agents to learn faster from a limited number of experiences. This speedup results from having twice as many gradients, generated from experiences of both agents, applied to the same set of shared parameters. Similarly, sharing parameters reduces the memory footprint of both the actor and critic networks since the total number of unique parameters is reduced. The savings in memory could subsequently be used to increase the number of nodes in each network.

One final consideration is whether to share parameters between agents for both the actor and critic networks or just one of the two. Only sharing actor parameters would enforce similar policies, but these policies could be pushed in different directions by the different critics. Sharing only critic parameters would help encourage similar gradients, but may not sufficiently constrain similarity between actor networks. We choose to share parameters for a given number of layers in both the actor and critic networks.

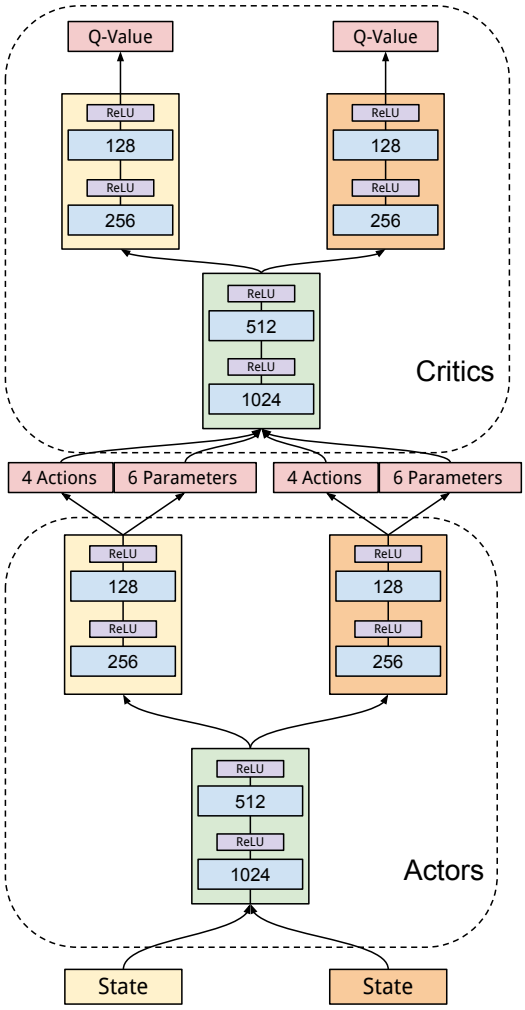


Figure 5.2: **The Parameter Sharing Architecture** shares the weights of a certain number of layers between the actor and critic networks of two or more agents. Sharing parameters encourages similar policies between the agents and can expedite the learning process due to the double gradients in the shared layers. This figure depicts two shared layers in both the actor and critic networks.

5.6 Memory Sharing

In addition to sharing parameters, we also examine the case in which agents learn from experiences selected randomly from a shared replay memory. In this approach, both agents share a single replay memory, both add their most recent experiences to the shared memory, and both perform updates from experiences selected at random from the shared memory. A shared memory encourages similar behavior from both agents since both learn from the same pool of experiences. As soon as a single agent learns to score a goal, both agents can leverage this experience to learn how to score. Since updates are performed from experiences selected at random, it's possible that the agents may randomly sample different experiences and learn different policies. However, in expectation, both of the agents will encounter many of the same experiences.

In the future it would be interesting to identify which experiences are most important for learning and transmit these experiences to the teammate. Something like the priority replay memory (Schaul et al., 2015) could be used to identify the most formative experiences of each agent.

In many ways, a shared replay memory has a similar effect to shared parameters. Both approaches encourage similar policies to be learned, but the parameter sharing approach is much more direct, since it enforces similarity by making the function approximators directly resemble each other. The shared memory approach is more indirect in the sense that similar policies are only derived by processing similar experiences. Both approaches encourage similarity, but whether or not similarity leads to cooperation depends on the task.

We have presented several multiagent architectures: a centralized control baseline, an architecture for sharing of parameters between the actor and critic networks, and a method for sharing experience replay buffers between different agents. The next section evaluates these architectures on a multiagent version of the soccer task.

5.7 Results: Multiagent Empty Goal Task

Experimentally, we compare the paradigms of independent learning, centralized control, parameter sharing, and replay memory sharing on the multiagent empty goal task (Section 5.1). Figure 5.4 shows the goal percentages and learning curves achieved by the independent learning and centralized control baselines. To summarize, both of these approaches result in exactly one agent learning to perform the task and the other agent showing no ability. In the case of independent learning, one agent, by chance, always learns how to perform the task before the other. This competent agent then maximizes its own rewards by continuing to score goals. The other agent has even less opportunity to learn the task since its teammate quickly moves the ball.

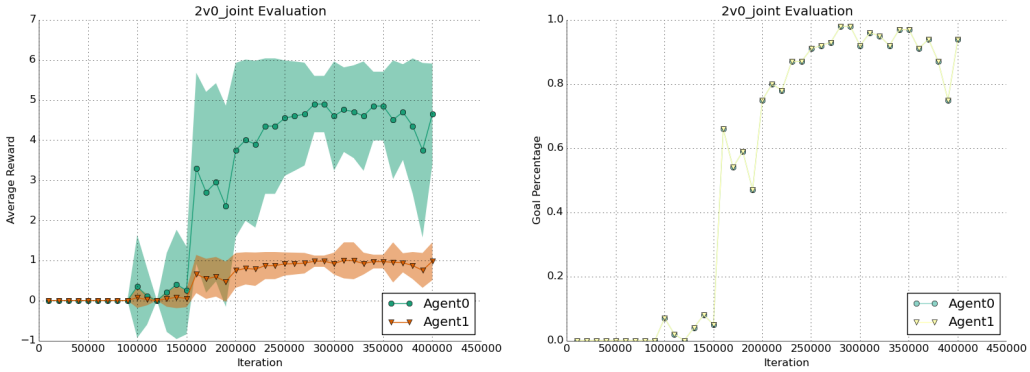
The case of centralized control is even more exaggerated: the centralized controller learns to use one agent exclusively for scoring goals, and learns to walk the second agent away from the ball entirely. Both of these controllers solve the underlying task, in the sense that goals are scored every episode, yet exhibit very little cooperation or coordination between agents.

Figure 5.5 depicts the performance of the agents utilizing the parameter sharing architecture. These agents share two layers in both the actor and critic networks. Similar to the independent learning case, one of the two agent learns to score before the other (e.g. before iteration 300,000). However, subsequently, the second agent gains competency in the task and both agents eventually end up sharing the rewards equally. In contrast to the independent learning baseline, in which the second agent could never learn to score, the shared weights from the first agent allow the second agent to also learn to score. Implicit in the actor network's shared weights was knowledge about an effective policy for scoring and in the critic's shared weights, better Q-Value estimation. Observing the resulting policies (Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/shareparams_2layer.mp4`) shows cooperative behaviors such as passing emerge between the two agents.

The memory sharing architecture, like parameter sharing, also results in both agents learning effective policies for scoring goals. Unlike parameter sharing, mem-

ory sharing features no delay between the time the first and second agents gain competency. Instead both agents track nearly the same learning curve and both are reliably scoring by 250,000 iterations. Note that the 250,000 iterations required to solve the task with the memory sharing approach is slightly slower than the parameter sharing approach, in which the first agent learned to score after only 150,000 iterations.

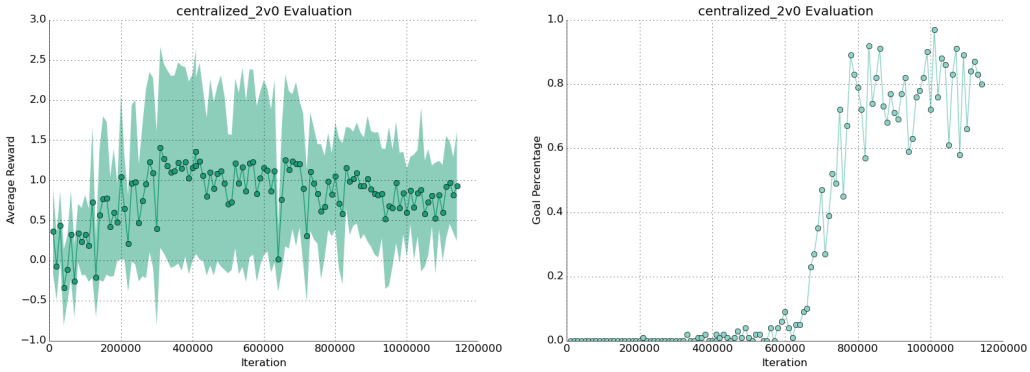
From these results we conclude in order to facilitate robust learning from both agents, it is necessary to have some type of sharing between the agents: both the parameter sharing and memory sharing approaches result in both agents learning to score goals rather than a single agent dominating the task. The next section examines the same approaches applied to the more difficult task of scoring on a goal keeper.



(a) Per-Agent Reward

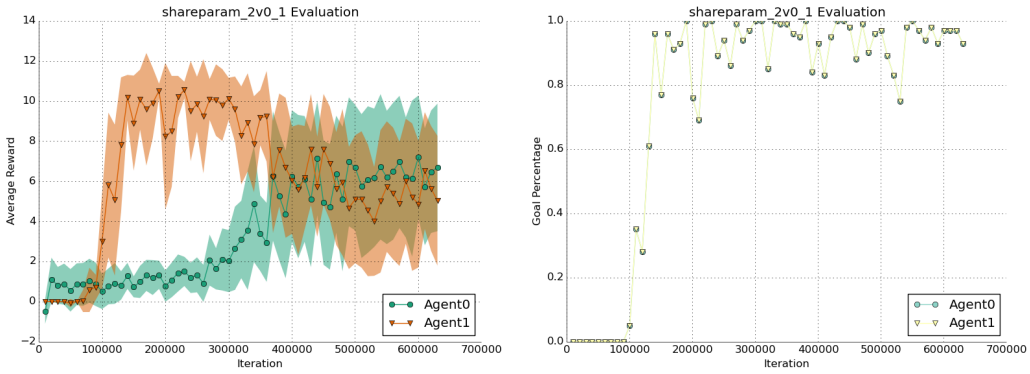
(b) Goal Percentage

Figure 5.3: **The Independent Learning**: baseline shows strong performance from Agent-0 who learns to fully perform the task by iteration 150,000. Agent-1 never learns to perform the task as a result of Agent-0 continually possessing the ball. Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/2v0_joint.mp4`



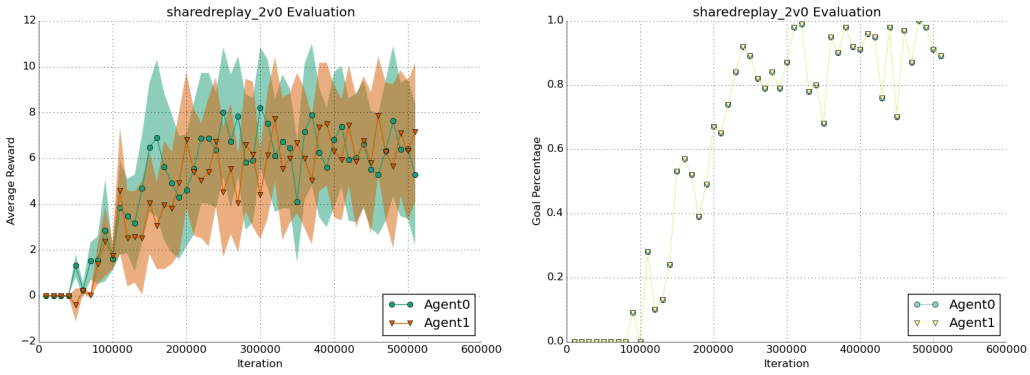
(a) Reward (b) Goal Percentage

Figure 5.4: **Centralized Controller**: The centralized controller learns to rely on one of the two agents to solve the task and ignores the second agent. Such behavior makes sense given the competitive nature of the reward function (Section 5.1). Note that the reward graph (left) shows the reward of the centralized agent which is a sum of the rewards achieved by each individual agent. Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/centralized_2v0.mp4`



(a) Per-Agent Reward (b) Goal Percentage

Figure 5.5: **Parameter Sharing** results in both agents learning to score goals and share the ball. In this experiment, the first two layers of the agents actor and critic networks were shared (as seen in Figure 5.2). As can be seen in the reward curves, Agent-1 first becomes competent at the task, but due to the shared weights, Agent-0 quickly also learns. As a result Agent-1's total reward is reduces when Agent-0 shares the goals. Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/shareparams_2layer.mp4`



(a) Per-Agent Reward

(b) Goal Percentage

Figure 5.6: **Replay Memory Sharing** results in both agents learning to complete the task and sharing in the resulting rewards. In contrast to parameter sharing, both agents in the replay memory sharing setting learn competency at the same time, rather than one learning first and transmitting knowledge (in the form of weights) to its teammate. Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/sharereplay_2v0.mp4`

5.8 Results: Multiagent Soccer vs. Keeper

As discussed in Chapter 4.8.4, learning to score on a keeper is a much more challenging task than scoring on an empty goal. Like the empty goal task, scoring on a keeper requires the offensive agents to locate, approach and dribble the ball. However, the offense agents now need to contend with a goal keeper using a hand-coded policy created by the Helios RoboCup team. The reward function used by the agents is identical to the reward function of the multiagent empty goal task (Equation 5.1), and provides rewards for approaching the ball, moving the ball to the goal, and scoring.

The benefit of this increased difficulty is an increased incentive for the agents to work together. Specifically, if the agents learn to pass the ball before shooting, they can effectively create a much larger goal opening than if a single agent was to approach the goal and shoot. The same tactic is often used in real soccer games, in the form of a cross kick.

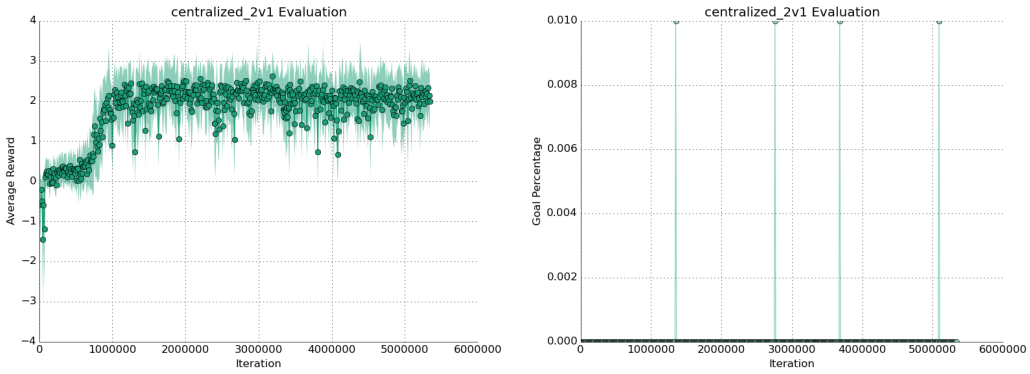
We analyze the independent learning, centralized control, parameter sharing, and memory sharing approaches on the 2v1 task. Experimentally, due to the increased difficulty of the 2v1 scenario, we provide 10-million training iterations rather than the 1-million used in the empty goal experiment.

As before, the centralized controller (Figure 5.7) learns to exclusively rely on one of the two agents to approach the ball and move the ball towards the goal. Reliable scoring is not achieved and the final policy simply dribbles the ball towards the goal, taking advantage of the reward provided for minimizing ball-goal distance, but then lets the ball be captured by the keeper.

Sharing parameters (Figure 5.8) results in both agents nearly solving the task. Examining the learned policies (Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/shareparam_2v1_1.mp4`), both agents demonstrate the capacity to approach the ball and dribble towards the goal. However, only Agent-0 learns to take shots on goal. Agent-1 dribbles the ball close to the goal but stops at a distance that prevents the goalie from capturing the ball. Next, Agent-1 waits for Agent-0 to approach the ball and shoot on goal. By waiting near the ball, Agent-1 ensures the episode does not end prematurely due to the 100-timestep limit of no agent touching the ball. The agents do not exhibit passing behavior or any more sophisticated coordination than holding the ball. However, even the holding strategy is effective enough to result in successful goals in roughly eighty percent of the episodes.

Finally, the memory sharing architecture (Figure 5.9) results in performance of both agents staying near identical throughout the learning process. Indeed, when observing the learned policies, it is apparent that both agents can effectively approach, dribble, and shoot on goal. Thus, in this regard, the agents are even more capable than in the parameter sharing case. However, coordinated passes or cross kicks are still notably absent and policies reliable enough to score every episode are not found. To summarize, the most coordination is observed from the replay memory sharing architecture, with the parameter sharing coming in a close second. In general, we expect that the memory sharing approach would be similar to a version of parameter sharing in which all the layers were shared between the actor

and critic. As before, the centralized controller fails to learn to coordinate the two agents.

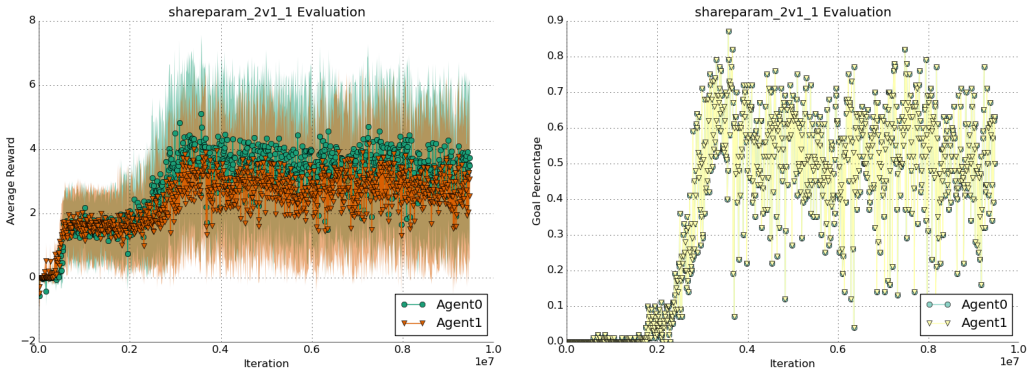


(a) Reward (b) Goal Percentage

Figure 5.7: **Centralized Controller (2v1)**: Like in the empty goal case, the centralized controller again learns to rely on only one of the two agents to handle the ball. This agent learns to approach the goal, but doesn't learn a robust policy for shooting. Note that the reward of the centralized agent (left) is the sum of the rewards of the individual agents. Agents are evaluated for 100 episodes and in the goal percentage graph (right), occasionally an agent will score one goal during the 100 episodes. Each goal corresponds to a peak in the graph. Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/centralized_2v1.mp4`

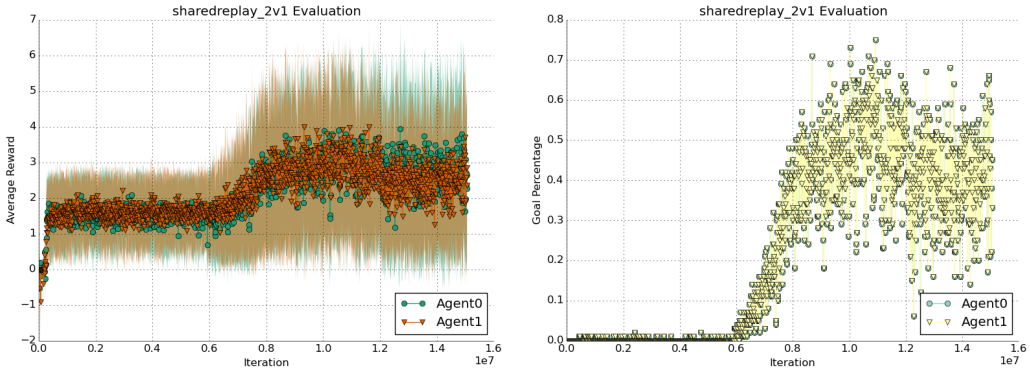
5.9 Analysis: Parameter Sharing

This section explores two aspects of parameter sharing: how many layers of the network to share and whether to share parameters between the actor network, critic network, or both. Specifically, Figure 5.10 shows results from the multiagent empty goal task when sharing a different number of layers in both the actor and critic networks. As the number of shared layers increases, the policies and rewards achieved by both agents become increasingly similar. When sharing four layers, the agents failed to robustly learn to perform the task, and score only occasional goals. These results suggest that sharing one or two layers results in the best performance.



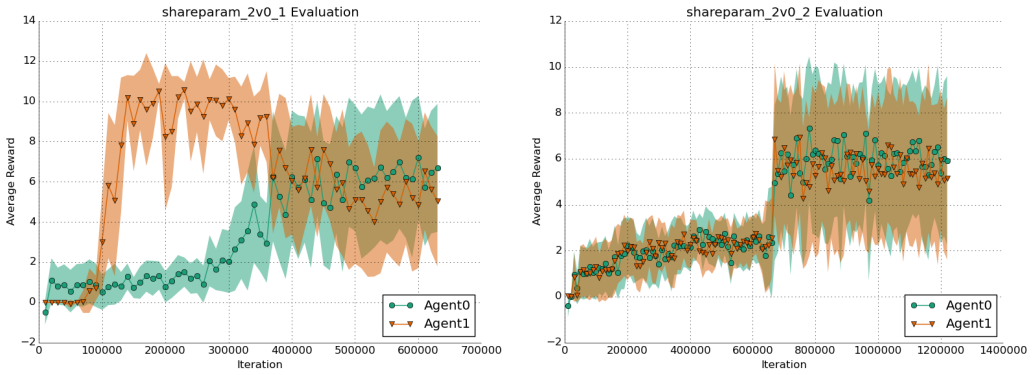
(a) Per-Agent Reward (b) Goal Percentage

Figure 5.8: **Parameter Sharing (2v1)** results in both agents learning to approach the ball and dribble towards the goal. However, only Agent-0 learns a policy for shooting on goal. Agent-1 assists by dribbling the ball towards the goal and keeping possession of the ball while Agent-0 is en route. Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/shareparam_2v1_1.mp4`



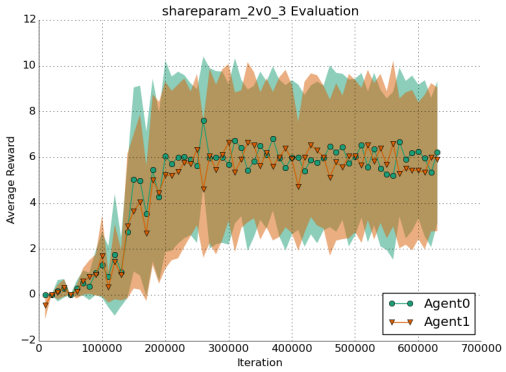
(a) Per-Agent Reward (b) Goal Percentage

Figure 5.9: **Replay Memory Sharing (2v1)** results in both agents learning to complete the task and sharing in the resulting rewards. In contrast to parameter sharing, both agents in the replay memory sharing setting learn competency at the same time, rather than one learning first and transmitting knowledge (in the form of weights) to its teammate. Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/sharedreplay_2v1.mp4`

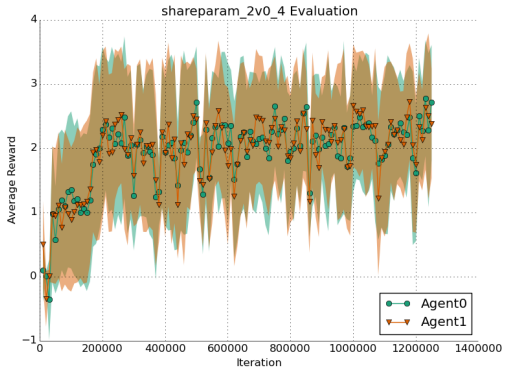


(a) Share 1-Layer

(b) Share 2-Layers



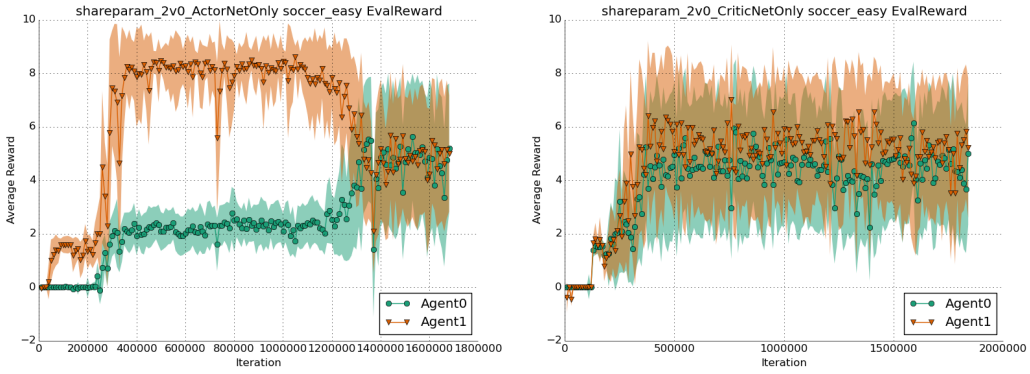
(c) Share 3-Layers



(d) Share 4-Layers

Figure 5.10: Number of shared layers influences similarity of policies. When sharing only the lowest layer of the network, the policies of the different agents can be significantly different. Sharing two or more layers results in very similar policies whose performance is highly coupled. The most robust goal scoring is observed when sharing only one or two layers.

We now compare sharing parameters between just the actor or just the critic network. For this experiment, we share the two lowest layers of the chosen networks and evaluate on the multiagent empty goal task. Results in Figure 5.11 show that sharing parameters in the actor network leads to less similar policies than sharing parameters in the critic network. Perhaps the best explanation for this phenomenon is that the weights of the actor network stem from the gradient updates originating from the critic network. Therefore, similar critics will lead to similar gradients and result in similar actor networks. In general, it seems to be a safe bet to simply share parameters between both the actor and critic networks.



(a) Share Actor Network

(b) Share Critic Network

Figure 5.11: Sharing parameters between the two lowest layers in only the actor networks results in dissimilar policies that require over 1-million iterations to both learn to score. Sharing parameters in critic networks results in more similarity between policies.

5.10 Chapter Summary

In general, multiagent reinforcement learning is a difficult proposition due to the challenge of discovering cooperative behavior and combating the non-stationarity introduced by time-varying policies from the different agents. Additionally, single agent deep reinforcement learning is not always stable in the sense that policies occasionally collapse or diverge. These difficulties are only compounded as the

number of learning agents increases. Nonetheless, this chapter demonstrates that cooperative multiagent learning is possible and analyzes approaches that facilitate stability and coordination between agents.

This chapter presented several different approaches for multiagent learning and evaluated them on two different multiagent tasks. At a high level, the intuition underlying many of the multiagent architectures is sharing. Sharing has two main advantages: First, if the agents can share parameters or experiences, they can bootstrap their learning not only from their own trajectories, but also their teammates'. Second, sharing parameters or experiences can help multiple agents achieve more stable policies by canceling out the perturbations inherent in a single agent's learning.

The nature of the learning task greatly influences the potential for coordination. We present two multiagent HFO tasks: coordinating with a teammate to score on and empty goal and coordinating with a teammate to score on a keeper. Both tasks heavily reward the agent that scores the goals, so a scoring agent has incentive to develop exclusive competency at the task. However, when sharing either parameters or replay memories, both agents learn to perform the task and occasionally exhibit cooperative behavior such as passing the ball.

In the case of scoring on a goal keeper, the sharing architectures again have top performance. However, no approach learns to perform this task perfectly and high-level coordinated plays such as cross kicks are lacking. Despite the fact that coordinated plays could increase performance of both agents, at least in the 2v1 task, we hypothesize that such behaviors are simply too hard to discover for our proposed architectures. For example, to perform a successful cross, both the kicking and receiving agents must correctly position themselves on the field. Next, the kicking agent must kick across the field, in a direction that the reward function will penalize. Finally, the receiving agent must be able to stop the ball and then kick on goal. The challenge of discovering and stabilizing such a cooperative behavior remains an open question.

In general, we are encouraged by the ability of sharing architectures to facilitate coordination between agents. The next chapter moves a step beyond sharing

and investigates active communication between agents.

Chapter 6
Communication

This chapter examines how deep reinforcement learning agents can use active communication to facilitate coordination. In order to get the most from this chapter, it helps to understand Chapter 4, particularly the architecture for learning in parameterized action space. Additionally, from Chapter 2, a basic understanding of backpropagation and deep neural networks is helpful. This chapter relates to thesis contribution 3: an exploration of multiagent deep RL in HFO.

The multiagent architectures presented in Chapter 5 were designed to facilitate coordination between multiple deep reinforcement learning agents. Indeed, in many of the domains, coordinated behaviors are found after tens of thousands of episodes of experience. However, in the real world, coordination between humans is often achieved far more quickly through spoken language. Humans can identify a problem that needs to be solved, communicate about how each member of a team should address the problem, and cooperatively solve the problem. Thus, when confronted with a problem, communication serves two main roles: first, communication helps to create a cooperative strategy for solving the problem. Second, communication may be further used as a part of this strategy while solving the problem.

Consider the example problem of Patrick moving to a new apartment. Patrick has a number of furniture items that he would like moved from his old apartment to his new one, and has invited a number of friends over to help with the moving process. First Patrick discusses which pieces of furniture he wants moved to the new apartment and which will be discarded. Next, Patrick and friends form small groups to move the heavy items to the new apartment. During the process of moving a heavy item, Patrick's friends must communicate to decide when to take rests, how to carry bulky items around corners, and which way to rotate the item to fit it through the door. This communication is accomplished through a shared language. This scenario highlights the importance of communication for solving cooperative

tasks: Patrick and friends first communicated a strategy for cooperatively solving the task, and then employed further communication when enacting that strategy. One can imagine the difficulty of moving if Patrick were unable to communicate which items he wanted moved or the location of his new apartment.

In the Half Field Offense environment, communication is a staple of organized team play. The HFO server allows each agent to send a single message of limited size every timestep. This message is heard by surrounding agents on both teams in the next timestep. Often, RoboCup teams will use obfuscated messages so their intent is hard to decipher by the opposing team. We will utilize this communication channel as a means to achieve further coordination between learning agents.

This chapter builds on the architecture presented in Chapter 4 for deep reinforcement learning in parameterized action space. In particular, communication actions are added to the agent's action space in the form of additional continuous actions. Each communication action transmits a single continuous value. Additionally, the state space of the agents is modified to include additional features for receiving incoming messages. In particular, the communication messages are continuous values and are transmitted, without modification, to the teammate on the next timestep, where they are concatenated with the agent's normal state features. The number of communication actions determines the bandwidth of the communication channel and is a hyper-parameter. In general, we expect that more channels have the potential to facilitate more complex communication, at the cost of increasing the size of the action space. The experiments in this chapter typically use either one or four communication actions. Figure 6.1 depicts the actor-critic architecture with added communication actions. We next present several approaches for using active communication to achieve cooperation.

6.1 Baseline: Independent Communication

The natural baseline is to allow the agents to use their communication actions as normal continuous actions and enforce no standardization or communica-

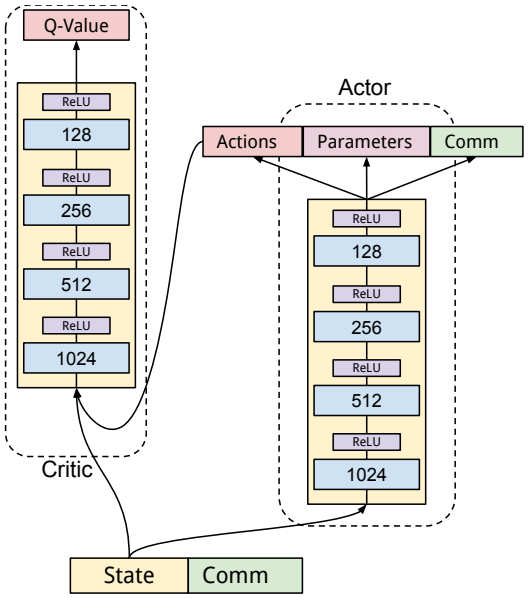


Figure 6.1: **Communication Architecture** appends continuous communication actions to the action space of the agent. Incoming messages received from other agents are concatenated with the agent's state space.

tion protocol. In essence, the agents may choose to transmit whatever messages they wish and pay attention to the incoming communication if they find it useful, or to entirely ignore the teammate's messages. We expect that communication of this form is likely to suffer from non-stationary induced by the changing policies of both agents. As a result, we hypothesize that it would be difficult for a stable and useful communication protocol to exist between the agents.

Relating to the work of Foerster et al. (2016b), the independent communication method is termed Reinforced Inter-Agent Learning (RIAL) because each agent uses independent reinforcement learning to determine the communicated messages.

6.2 Teammate Communication Gradients

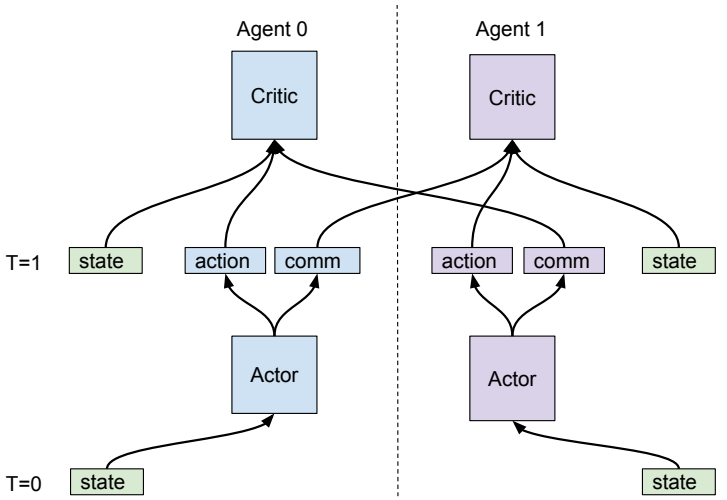
When humans encounter a new task, they often create new vocabulary to compactly describe important characteristics of the task. Nearly every hobby, skill, or sport has its own specialized vocabulary. For example, in the sport of rock climb-

ing, there are special terms to describe different rock formations, body positions, hold types, etc. This new vocabulary is developed and refined over the lifetime of the activity. Often, new vocabulary is driven by the need to quickly describe elements of the real world that are difficult to capture using only existing vocabulary. It is this drive that fosters the development of new vocabulary or alteration of the existing protocol.

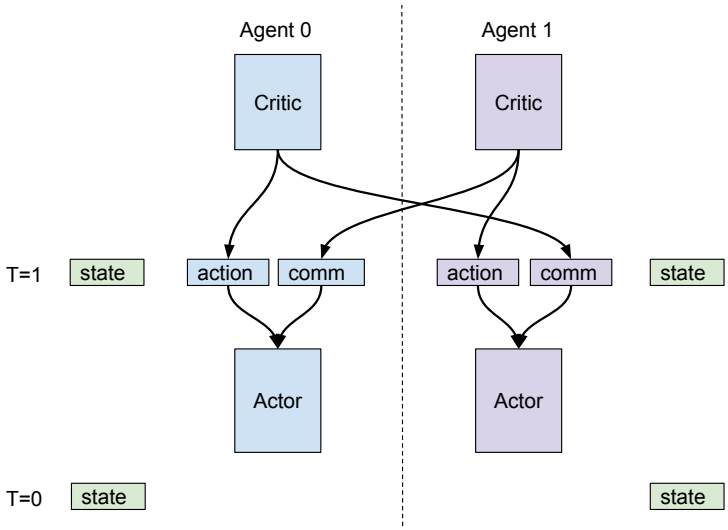
The teammate communication gradients approach mirrors this process: at a high level, this update allows agents to influence the communication strategy of their teammates through shared gradients applied to the teammates' communication actions. These gradients inform the teammate about how it can alter its messages in the direction of increasing rewards. This process is loosely analogous to developing new vocabulary or altering an existing communication protocol to solve a new task.

As shown in Figure 6.2, the teammate communication gradients update consists of two main steps: first, each agent's critic network computes a gradient specifying how the communication actions of the teammate should change. Second, these gradients are passed between agents and each agent's actor network is updated using the gradients supplied from the teammate. Note the communication update only applies to communication actions, and normal actions are still updated according to the standard actor-critic update described in Chapter 2.7.

Because teammate communication gradients are directly shared between agents, the typical boundary that divides independent agents is broken. However, it is only necessary to break this barrier during the training phase. At evaluation time, the agents act fully autonomously and use only the official HFO communication channels. In this way, the process of sharing gradients can be considered centralized training, followed by a decentralized execution. This is a common paradigm in RoboCup: teams will program a shared communication protocol when preparing for the competition and use the pre-developed protocol at competition time to send messages.



(a) Forward Pass



(b) Backward Pass

Figure 6.2: **Teammate Communication Gradients Update:** In the forward pass, the communication actions taken by each agent are concatenated with the state of the teammate in the next timestep. The critic evaluates the current state, the agent's action, and the teammate's message. In the update, each agent's current communication action replaces the one found in the replay memory. The backwards pass modifies the actor network's action in accordance with the agent's own critic network. However, the communication actions are modified in accordance with the teammate's critic network.

6.3 Grounded Semantic Network (GSN)

Semantics is the branch of linguistics and logic concerned with meaning. Ultimately, the agents need to establish a meaningful communication protocol grounded in the state, action, and reward functions of the task. Such a protocol should answer the questions of 1) what to communicate about, and 2) how to communicate it. We expect that the necessary things to communicate will vary from task to task and an ideal approach should be able to adapt the communication protocol to the task at hand.

The grounded semantic network (depicted in Figure 6.3) fulfills these criteria by learning a model of the teammate's one-step reward, conditioned on the teammate's action and the agent's state-observation. The hidden-layer activations of this model are communicated to the teammate as a message. The communication protocol learned by the GSN is grounded in the observation space and reward function of the task and embodies a semantic mapping: a transformation from concept to message.

From the perspective of Agent-1, a GSN learns a mapping from the observation $o^{(1)}$ and teammate action $a^{(2)}$ to teammate reward $r^{(2)}$. The network contains two major parts: a message extractor $m^{(1)} = M(o^{(1)}; \theta_m)$ which maps the agent's observation into a message, and a one-step reward model $\hat{r}^{(2)} = R(m^{(1)}, a^{(2)}; \theta_r)$ that predicts the teammate's immediate reward. Composing these components, the GSN computes the following function:

$$\hat{r}^{(2)} = R\Big(M\big(o^{(1)}; \theta_m\big), a^{(2)}; \theta_r\Big) \tag{6.1}$$

GSN training follows a supervised learning paradigm. Given experience tuple $(o^{(1)}, a^{(2)}, r^{(2)})$, the GSN is trained to regress its predictions towards the rewards of the teammate, minimizing the following loss function:

$$L(\theta_r, \theta_m) = \mathbb{E}_{(o^{(1)}, a^{(2)}, r^{(2)})}\left[\Big(r^{(2)} - R\big(M(o^{(1)}; \theta_m), a^{(2)}; \theta_r\big)\Big)^2\right] \tag{6.2}$$

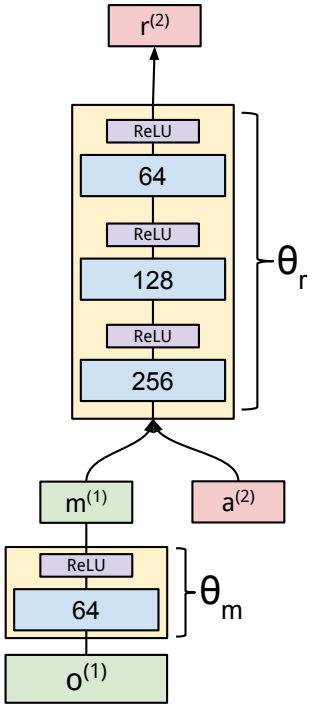


Figure 6.3: **The Grounded Semantic Network** predicts the teammate's one-step reward $r^{(2)}$ conditioned on the agent's current observation $o^{(1)}$ and teammate's action $a^{(2)}$. The message $m^{(1)}$ is an intermediate layer in this network, and learns a compact representation of the current observation that is useful for predicting teammate reward. The activations of the message layer are transmitted to the teammate as the message. Training a GSN requires direct access to the teammate's actions and rewards. However, at test time, only the agent's observations are required to generate the message.

Similarly to teammate communication gradients, GSN follows a centralized training procedure with decentralized execution: during training, GSN requires direct access to the teammate's actions and rewards. Direct access breaks the standard boundaries between independent agents. However, at execution time, only the agent's current observation is required to generate a message.

6.3.1 Stability

In the context of deep reinforcement learning, experience tuples are generated from interactions with the environment, stored in a replay queue, and sampled randomly for updates. Unlike standard supervised learning from a fixed dataset, there is a dangerous loop in which the policies of the agents affect the messages generated by the GSN, which, in turn, affect the policies of the agents. Without care, such a loop can result in instability or collapse of the communication protocol and the agents' policies. Such failures were observed in preliminary experiments. To alleviate this danger, we train the GSN with a learning rate of 10^{-6}, an order of magnitude smaller than the learning rate used to train the agents' policies. This reduced learning rate encourages slow changes to the GSN and allows the agents to smoothly adapt to alterations in the communication protocol.[1]

6.3.2 Limitations

Intuitively, the GSN combats partial observability of the multiagent environment by learning a transformation of the agent's observation that is relevant for predicting the teammate's reward. In essence, if there is information available in Agent-1's observations that could help Agent-2's performance, the GSN will learn to extract and communicate this information. However, the GSN is not conditioned on Agent-1's policy and cannot extract the intentions of Agent-1. The abilities and limitations of the GSN are explored further in the experiments below.

The next sections evaluate the approaches of independent communication,

[1] No systematic stability gains were observed from using target networks or reducing the ratio of GSN updates versus policy updates.

teammate communication gradients, and GSN on two cooperative multiagent domains.

6.4 Results: Say My TID Task

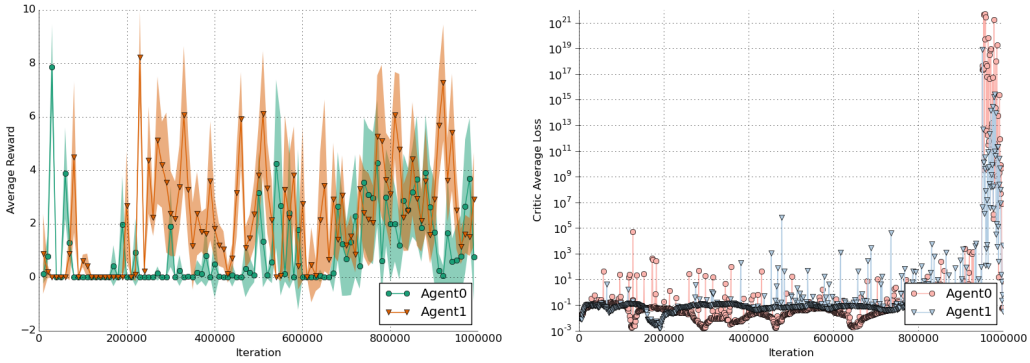
Say My TID is a two-player game in which each agent is assigned a secret number (or Thread Identifier - TID), represented by a single floating point value. The goal is for each agent to help its teammate correctly guess its hidden number. This domain uses a single communication action, so each agent is allowed to send one floating point value every timestep. Both agents are rewarded for minimizing the distance between the teammate's message $m \in \mathbb{R}^1$ and their own hidden value h. Specifically, reward for Agent-1 is $r_t = \alpha / e^{\beta(h - m_{t-1})^2}$, where m is the message sent by the teammate in the last step, $\alpha = .1$ and $\beta = 50$ are constants controlling the magnitude and decay of reward. Reward is symmetric for Agent-2. Because episodes are constrained to last at most 100 timesteps, the maximum achievable reward is 10. Since the secret number of each agent is hidden from the teammate, this task can only be solved by the agents communicating which messages are correlated with high reward.

Examining the results, the independent communication baseline is unable to solve this task. Since the independent communication approach lacks a way of establishing a stable communication protocol, it is not surprising that independent communication agents cannot solve this task. Figure 6.4 shows that in the process of exploration, both agents do, at times, get close to saying the TID of their teammate. However, since each agent is trying to maximize its own rewards and has no established protocol for asking the teammate to communicate specific values, there are no signs of convergence or learning.

In Figure 6.5, by harnessing the extra communication channel, agents using the teammate communication gradients approach can find stable policies that maximize reward. This is possible because the teammate gradient update allows each agent to influence the messages that its teammate utters. Indeed, following the communication gradients provided by the teammate results in quick convergence to

the optimal solution.

Say My TID is an instance of a class of domains in which rewards correspond only to content of communicated messages, rather than interactions with the environment. Shown in Figure 6.6, these domains highlight a limitation of GSN: the inability to directly alter communication following a reward gradient. In such domains, DIAL remains the method of choice since it can directly alter the content of messages in the direction of higher rewards. The next section explores a more complex task.



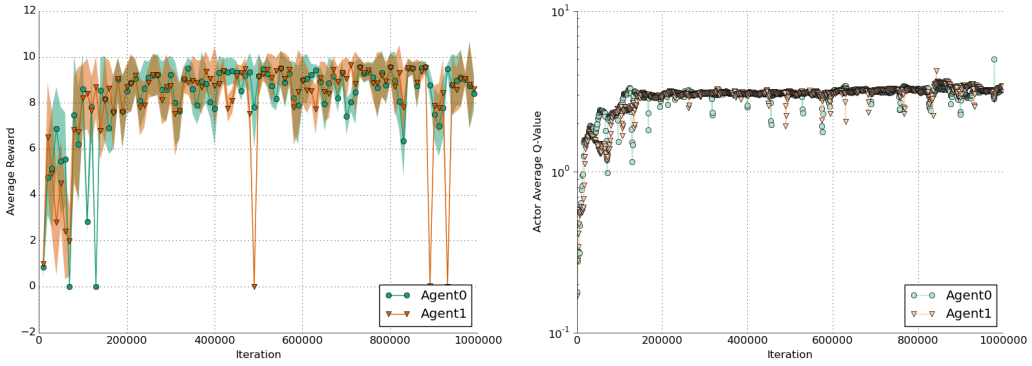
(a) Learning Curve

(b) Update Loss

Figure 6.4: **Independent Communication fails to solve Say My TID Task.** Left: there is no systematic increase in reward for either of the agents. Right: Neural network loss remains stable until the end of the learning process, when instability is observed. The failure to learn follows from the lack of an established protocol for communicating the needs of teammates.

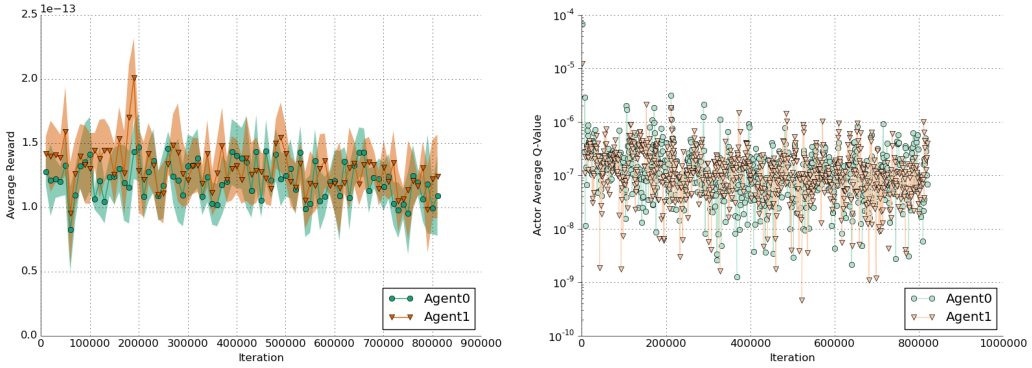
6.5 Blind Move to Ball Task

The goal of the Blind Move to Ball task is for a sighted agent to steer a blind agent towards the ball using only communication. This task is performed by a blind agent and a sighted agent. The blind agent cannot see anything on the field: its normal state features are present but constantly zeroed. It can only hear incoming communication messages, which are appended to its state and are not zeroed. The



(a) Learning Curve (b) Update Loss

Figure 6.5: **Teammate Communication Gradients solves Say My TID Task.** Left: reward quickly increases to the maximum achievable reward of 10. Right: Update loss remains stable throughout the learning process. Success is achieved by each agent altering the messages of its teammate towards higher individual reward.



(a) Learning Curve (b) Update Loss

Figure 6.6: **GSN fails to solve Say My TID Task.** Because GSN is trained to predict rather than maximize teammate reward it cannot solve the Say My TID task. GSN learns messages correlated with teammate rewards rather than messages maximizing teammate reward.

sighted agent has normal observations but cannot move (specifically it can still turn but cannot dash). Both agents are rewarded for minimizing the distance between the blind agent and the ball: $r_t = d_{t-1}(a, b) - d_t(a, b)$. At the start of each episode, the blind agent, teammate, and ball are initialized randomly on the field. Episodes end when the blind agent reaches the ball or 100 timesteps pass. Since the blind agent cannot see the ball, the only way to solve this task is for the sighted agent to learn a stable protocol for directing the blind agent towards the ball. It is impossible for either agent to solve the task alone or without communication.

6.6 Results: Blind Move to Ball

The independent communication baseline in Figure 6.7 reaches moderate performance levels, but lacks stability to maintain them. We suspect this lack of stability results from non-stationary communication protocols. The teammate communication gradients approach completely fails to solve the Blind Move to Ball task using one, two, or four communication actions. As shown in Figure 6.8 no learning is seen, despite having one million iterations to find a stable communication protocol. Finally, as shown in Figure 6.9, GSN learns a communication protocol capable of solving the Blind Move to Ball task. To further understand this result, we analyze the details of the approaches.

The downfall of the teammate communication gradients approach is that the communication gradients are not grounded in reality. Specifically, the blind agent communicates what it wants to hear (via communication gradients), but that does not mean what it wants to hear actually reflects reality. For example, the blind agent wants to hear that the ball is directly in front of it, because it can easily obtain reward from dashing forward. So the blind agent shapes the teammate's messages to always say that the ball is directly ahead. However, in reality, the ball may or may not actually be directly ahead of the blind agent, and as seen in the results, a communication protocol not grounded in reality has little practical value. The teammate communication gradient approach works well on domains such as Say My TID where reward stems directly from the communicated message. In such

cases, it is only necessary to optimize the content of the messages, and communication does not need to be grounded in reality. In contrast, GSN learns a communication protocol that is correlated with teammate reward and remains grounded by the actual state of the environment. GSN performance is best on domains in which communication is used as a means to achieve a goal in the environment.

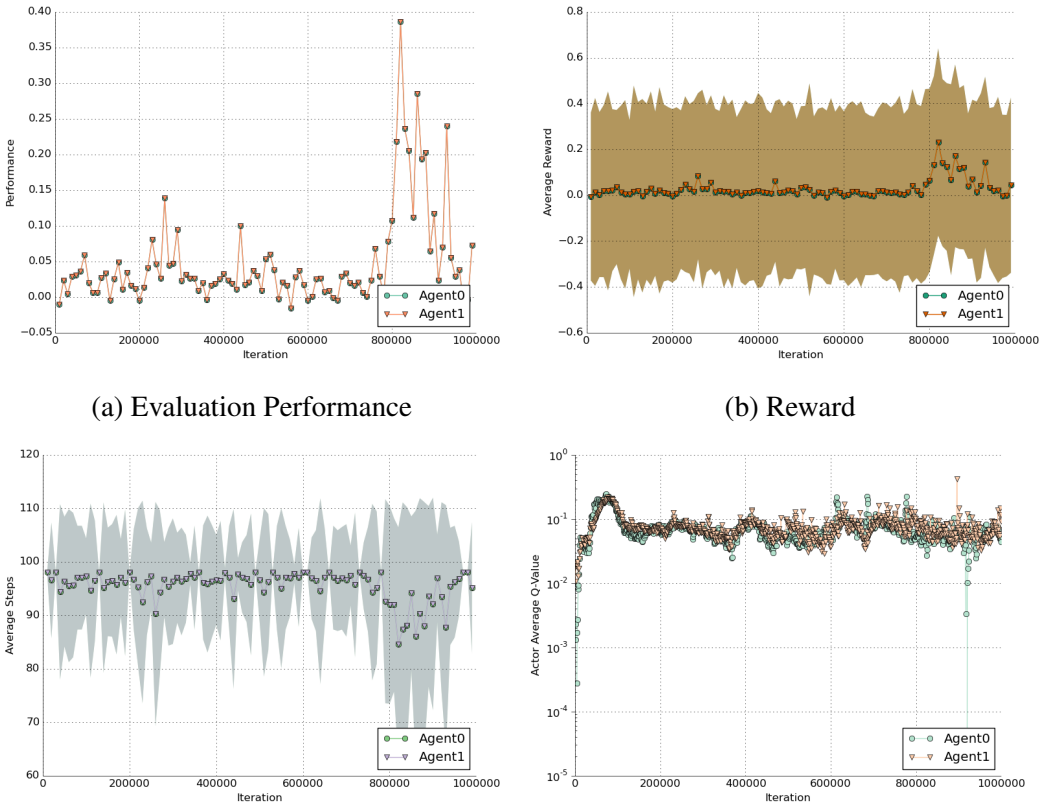
It should be noted that the normal move to ball task is typically solved by a sighted agent within 50,000 iterations, compared to the 400,000 iterations required by the GSN to solve the Blind Move to Ball task.[2] However, the GSN faces two major challenges that collectively explain the longer learning times. First, GSN must bootstrap a communication protocol while also bootstrapping policies for both agents. Second, a communicated message features a one step delay before it is received. So even a blind agent using a perfect communication protocol would not be as capable as a sighted agent, since the sighted agent does not suffer any delays in information. For reference, a sighted agent with a single step state delays takes 200,000 iterations to master the move to ball task: four times as long as the agent with no state delay, but still twice as fast as the GSN agent.

6.7 Analysis

We perform an ablation analysis on policies learned in the Blind Move to Ball task by disabling communication and re-evaluating the learned policies of each agent. Policies learned by independent communication and teammate communication gradient approaches remain unchanged when communication is disabled, indicating that communicated messages are not actively used by the blind agent. In contrast, GSN's policy is adversely affected by a lack of communication: without guidance from the sighted agent, the blind agent walks directly forward regardless of the location of the ball Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/GSN_MoveToBall_NoComm.mp4`.

In order to further analyze the communication protocol learned by GSN on the blind soccer task, Figure 6.10 visualizes the space of messages sent by the

[2] See Figure 7.3 for single agent performance.



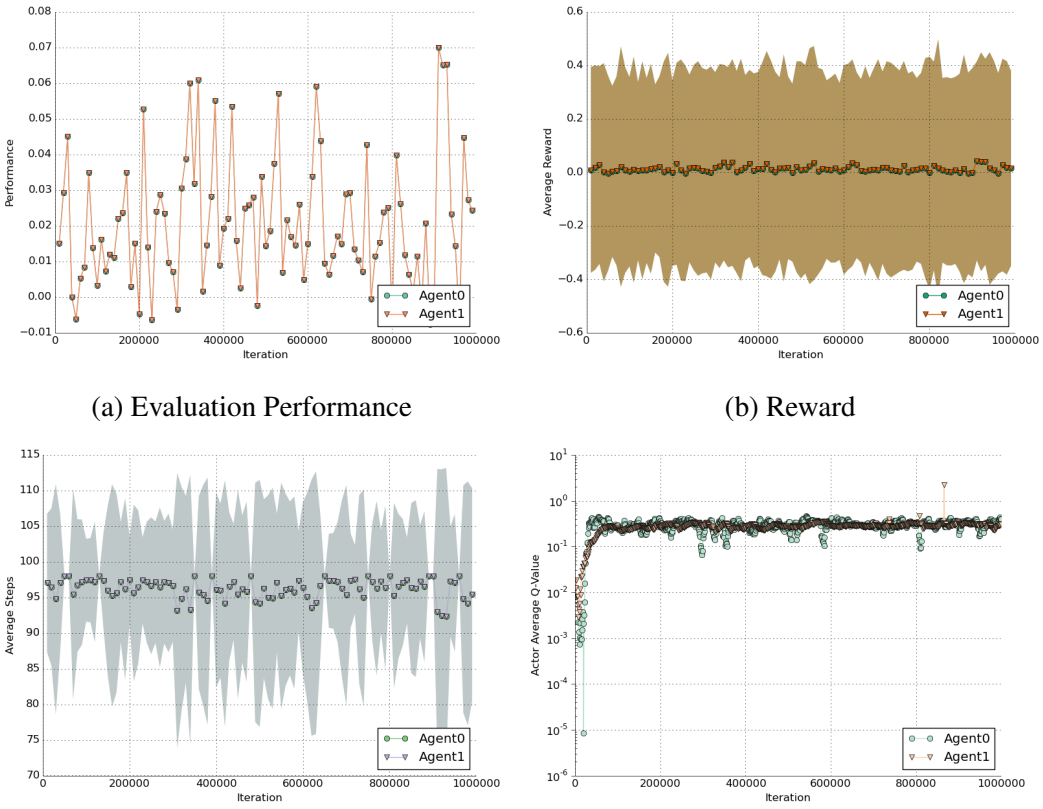
(a) Evaluation Performance

(b) Reward

(c) Steps per Episode

(d) Actor Average Q-Value

Figure 6.7: **Independent Communication Baseline fails to solve the Blind Move to Ball task.** Using 4 communication actions, some evidence of learning is observed, but the resulting policy is not stable and cannot maintain high performance. This instability is likely caused by the changing nature of each agent's communication actions. Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/BlindMTB_CommAct4.mp4`



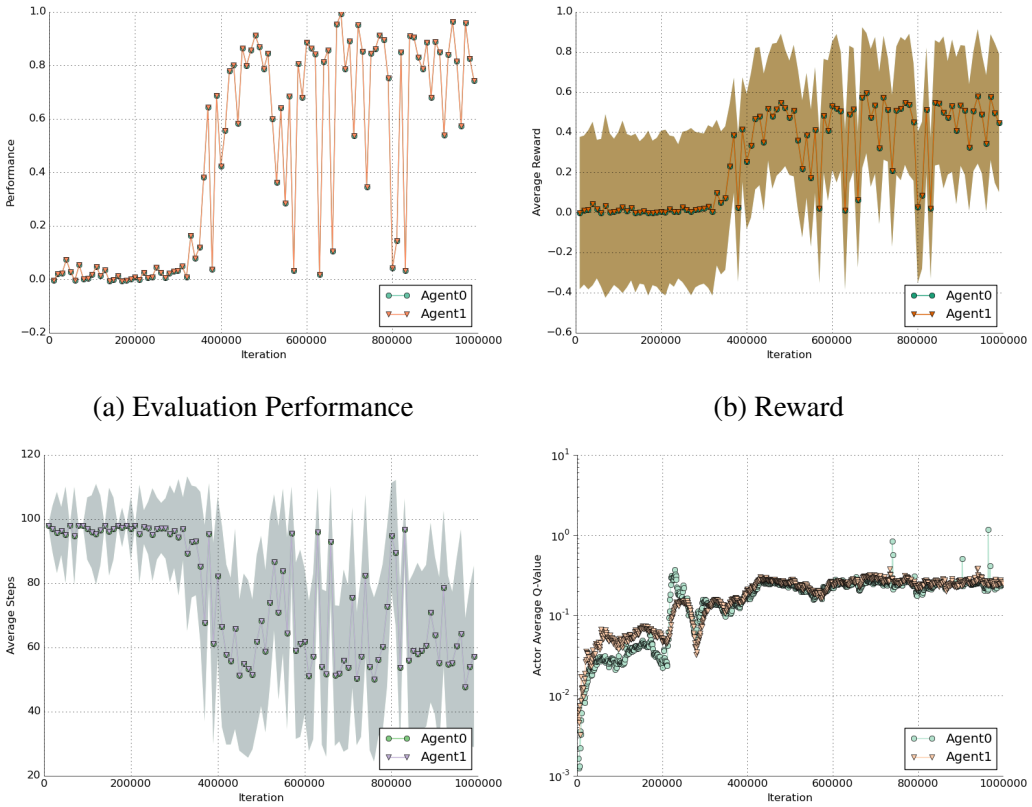
(a) Evaluation Performance

(b) Reward

(c) Steps per Episode

(d) Actor Average Q-Value

Figure 6.8: **Teammate Communication Gradients fails to solve the Blind Move to Ball task.** This particular experiment uses 4 communication actions. No evidence of learning is observed in this experiment or in similar experiments using 1 or 2 communication actions. The policy is stable and the learning process does not experience divergences or collapse, but is simply unable to learn. Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/BlindMTB_CommAct1_ApproxGrad.mp4`



(a) Evaluation Performance

(b) Reward

(c) Steps per Episode

(d) Actor Average Q-Value

Figure 6.9: **GSN solves Blind Move to Ball task:** GSN achieves the maximum reward of 0.6. Note the variance of rewards is high because total reward is proportional to the distance between the blind agent and the ball. Random initialization of agents and ball on the field results in high variance of reward even for a perfect agent. Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/GSN_MoveToBall.mp4`

sighted agent using t-SNE (van der Maaten and Hinton, 2008). As shown in the figure, there is a strong correlation between the content of the message and the action selected by the teammate in the next timestep. This correlation illustrates that messages contain information that is useful for the blind agent to decide what high-level action it should select. Qualitatively, this shows that the GSN has correctly found a communication protocol that is useful for the task.

Figure 6.10: **t-SNE Visualization of Communicated Messages:** t-SNE shows a 2-dimensional projection of 4-dimensional messages sent by the sighted agent while performing the Blind Move to Ball task. Messages with similar content are mapped close to each other in the figure. Each message is colored according to the action taken by the blind agent in the next timestep: black dots correspond to Dash actions and white dots are Turn actions. The fact that black dots are clustered in one region of the space and white in the other shows that content of the sighted agent's messages is correlated with the actions selected by the blind agent.

Finally, it helps to visualize the messages and the policy at the same time. Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/gsn_vis.mp4` shows a t-SNE visualization of messages being created as the agents perform the task. From this video, it is clear that the messages indicate how far the blind agent is from the ball, as well as whether to turn or dash.

6.8 Related Work

There is an extensive body of literature on communication between reinforcement learning agents (Tan, 1998; Panait and Luke, 2005; Zhang and Lesser, 2013).

Stroupe et al. (Stroupe et al., 2001) show that a blindfolded robot is able to track a moving soccer ball by using the observation of two sighted teammates. This work reinforces the idea that multiple agents can overcome partial observability by fusing distributed sensor readings. However, these robots rely on a pre-established rather than learned communication protocol.

Kalyanakrishnan has explored the communication of values between agents (Kalyanakrishnan et al., 2007). In particular, his agents transmit experiences to each other, similar to a selective version of memory sharing (Section 5.6). These messages include the full state of the agent, the action selected, and the rewards received.

Recently there have been several advances in deep multiagent reinforcement learning. Foerster et al. (2016a) describe an approach used to train multiple recurrent Deep Q-Networks to solve two riddle domains. The authors describe three alterations to standard DQN that were necessary for robust learning: first they provided each agent's previous action as input to the next timestep. Second, they shared the parameters between the networks of each agent - so in effect, only one network is learned by all the agents. Third, they found it necessary to disable experience replay. With these modifications, they show positive results on two well-known riddles, where the discrete action space of the agent involves communicating with the other agents. These results are encouraging as they show that multiple agents can learn a stable protocol for performing cooperative tasks.

Extending this work, Foerster et al. (2016b) explores additional approaches for learning communication between agents. The two approaches presented are Reinforced Inter-Agent Learning (RIAL) and Differentiable Inter-Agent Learning (DIAL). RIAL is similar to the previous work in that two DQN agents share network parameters and learn to communicate simply by interacting with the environ-

ment. On the other hand, DIAL, involves not only sharing network parameters but also sharing communication gradients between the agents. The DIAL paradigm involves a centralized learning phase where communication gradients assist the discovery of a stable communication protocol, followed by a decentralized execution phase. These approaches are validated on the same two riddles used in the last work and it is shown that DIAL is able to reach optimal performance faster than RIAL, indicating that fully-differentiable communication is a benefit for multiagent learning. The communication architectures presented in Chapter 6 draw on similar ideas to RIAL and DIAL. The main difference is that communication in this thesis takes places over a continuous rather than discrete communication channel, eliminating the need for a discretize/regularize unit. Additionally, our experiments focus on solving problems in which communication is only a small part of the larger multiagent task and the agents need to learn a significant amount of task-based skills before communication can even be successfully leveraged. We hypothesize that real world domains are similar in that individual competency is a prerequisite to fruitful cooperation.

6.9 Chapter Summary

This chapter was predicated on the idea of using communication to achieve greater cooperation between agents. To investigate this idea, we modified the action space of HFO agents to include one or more continuous communication actions. Using these communication actions, agents can transmit real-valued messages to their teammates. However, there is no preset communication protocol specifying what should be transmitted or what a message means. Instead, the agents must learn how to communicate effectively with each other in order to solve a task.

We presented several methods for learning communication: the independent communication baseline treats communication actions the same way as standard actions and uses gradients generated from the critic network to alter the communicated messages. Teammate communication gradients is an approach for sharing communication gradients between agents and allows each agent to alter the mes-

sages sent its teammate in the direction of higher rewards.

Finally, the Grounded Semantic Network is a trainable model that learns a task-dependent communication protocol for solving cooperative multiagent tasks. We introduced and evaluated the GSN and competing approaches on two domains - the Say My TID task rewarded optimizing the content of messages, and the Blind Move to Ball task used communication as a means to solve a guide a blind agent towards a soccer ball. GSN outperforms the other two approaches on the Blind Move to Ball task. Analyzing the communicated messages shows that the communication protocol is highly correlated to the actions selected by the blind teammate. In general, these results highlight the ability of deep reinforcement agents to learn a communication protocol that overcomes partial observability and help facilitate cooperation between independent agents.

Chapter 7
Curriculum Learning

Many of the tasks presented in earlier chapters have relied upon hand-designed reward functions that provide the agent with frequent, informative rewards. However, in HFO, true rewards should correspond to winning soccer games, or scoring goals. In practice, this type of reward function provides far too infrequent rewards for Deep RL agents to boostrap a policy from scratch. This chapter focuses on curriculum learning as an approach towards using sparse, unbiased reward functions. To understand this chapter, it is necessary to understand the basics of reinforcement learning and deep learning presented in Chapter 2. Additionally, familiarity with the actor-critic architecture (Chapter 4) for learning in parameterized action space is helpful. This chapter addresses thesis contribution 5: curriculum learning in HFO.

At a high level, this chapter presents curriculum learning approaches used to divide the complex task of playing soccer into smaller, more manageable subtasks of approaching the ball, kicking the ball to the goal, dribbling, and passing. In order to learn the complex original soccer task, the agent must first learn a curriculum of easier subtasks. To this end, we first present a motivating discussion that illuminates the difficulty of hand-designing unbiased reward functions. Building on this motivation, we present related work that has addressed the problem of learning in complex domains with sparse reward signals, and discuss the curriculum approach we used to tackle the soccer task with on a sparse goal-based reward function.

7.1 On the Design of Reward Functions

True, unbiased rewards in the HFO domain come from scoring or defending goals. However, using only goal reward results in learning agents not seeing even a single positive reward. Conceptually, to score a goal, the agent must approach the ball and kick towards the goal. In practice, a randomly exploring agent never makes it to the ball, much less kicks towards the goal. In other words, such a reward

signal is far too sparse for learning agents to gain traction. Instead, in Chapter 4.1, to solve the single agent soccer task we introduced a hand-crafted reward signal with four components: **Move To Ball Shaping-Reward** provides a scalar reward proportional to the change in distance between the agent and the ball $d(a, b)$. An additional shaping-reward $\mathbb{I}^{kick}$ of 1 is given the first time each episode the agent is close enough to kick the ball. **Kick To Goal Shaping-Reward** is proportional to the change in distance between the ball and the center of the goal $d(b, g)$. An additional true reward is given for scoring a goal $\mathbb{I}^{goal}$. A weighted sum of these components results in a single reward that first guides the agent close enough to kick the ball, then rewards for kicking towards goal, and finally for scoring. The reward function (reproduced from Equation 4.1 in Chapter 4) is:

$$r_t = d_{t-1}(a, b) - d_t(a, b) + \mathbb{I}_t^{kick} + 3\big(d_{t-1}(b, g) - d_t(b, g)\big) + 5\mathbb{I}_t^{goal} \tag{7.1}$$

It is disappointing that reward engineering is necessary. However, with a sparse reward function the exploration task proves far too difficult because acting randomly in the low-level action space is exceedingly unlikely to yield even a single goal. The hand-engineered reward function provide a figurative trail of bread-crumbs: enough information to allow the agent to bootstrap its policy from random actions all the way to scoring goals. However, a major concern is that the shaping rewards could result in a learned policy which is suboptimal with respect to the true task of scoring goals.

7.2 Limitations of Potential-Based Shaping Rewards

Ng et al identify a class of potential-based shaping reward functions which guarantee that optimal policies found in the reward-shaped MDP are still optimal in the original MDP (Ng et al., 1999). They prove that any potential-based shaping

reward $F(s, a, s')$ is both necessary and sufficient to preserve optimality:

$$F(s, a, s') = \gamma\Phi(s') - \Phi(s) \tag{7.2}$$

Where $\Phi(s)$ is a an arbitrary potential function of the state. In Equation 7.1, **Move To Ball** and **Kick To Goal** components satisfy these requirements since they are differences of potential functions over consecutive states. Since the true task reward is the reward for scoring a goal, the only component of the reward function not guaranteed to preserve optimality is the shaping reward given for getting close enough to kick the ball.

In this light, it would be ideal if we could leverage shaping rewards to aid in the design of unbaised reward functions. Unfortunately, the theoretical guarantees given by potential-based shaping functions do not hold for the HFO domain and instead result in biased policies.

Ng et al. (1999) claims in Remark 2 that "All policies are optimal under a potential-based shaping function." This statement is true when there exists a single terminal state and episodes only end when the terminal state is reached. In such a situation, any policy that reaches the terminal state will have collected the same amount of potential reward.

However, in HFO, episodes end in a variety of ways: scoring a goal, running out of time, or ball being captured by defense. This multitude of terminal states violates the claim above. Consider the **Move To Ball** shaping reward, which is a difference of potential functions. A policy that moves the agent away from the ball ends the episode by running out of time and receives negative reward, while the policy that moves the agent toward the ball ends the episode with positive reward when time expires. Clearly, it is not the case that all policies are optimal under the move to ball potential-based shaping function.

On the other hand, if HFO episodes were allowed to run for an infinite amount of time and only terminated when the agent reached the ball, then any policy that eventually reached the ball would be optimal, and the move to ball shaping reward would be unbiased. To summarize, potential functions look like an enticing framework for designing unbaised reward functions, but are inapplicable to domains

featuring multiple terminal states, such as HFO.

The suboptimality of the hand-designed reward function in Equation 7.1 becomes more apparent in the multiagent case when two agents are tying to coordinate to score goals. If both agents use the reward function in Equation 7.1, one agent learns to approach the ball, dribble, and score goals. The other agent learns to follow directly behind the first, collecting move to ball and kick to goal rewards in the process. Specifically, the following agent is still rewarded for its teammate kicking the ball toward the goal and scoring. This following behavior would be suboptimal in the presence of an opposing keeper, since the following agent is badly positioned to receive a pass or take a shot on goal. For this reason, the multiagent task (discussed in Chapter 5.1) uses a hand-designed reward function that rewards only the agent on the ball. However, even this reward function has limitations. Namely, it does not provide rewards/encourage agents to pass or position themselves to receive passes.

In general, hand-designing reward functions becomes more complicated in the multiagent case: How much reward should be given for a teammate scoring versus the agent? Should an agent be encouraged to move away from the ball if its teammate is already approaching it? Should an agent be rewarded if its teammate moves the ball towards the goal? How should agents be rewarded for passing? How much suboptimality will be introduced by a mistake in weighting these different rewards? In many senses, hand-designing a reward function for a complex task can be as difficult as hand-coding a policy to solve that task. Therefore, this chapter presents approaches capable of learning from the sparse goal reward. The next section discusses related approches for handling sparse rewards.

7.3 Related Work

Complex tasks such as those with sparse rewards require more sophisticated approaches than plain DQN or DDPG. DQN and DDPG rely on frequent, if small, rewards in order to bootstrap an effective policy. If rewards are too sparse, the exploration policies used by DQN/DDPG will be unable to reliably encounter any

positive reward and will be starved of the gradients needed to bootstrap a good policy. There are a variety of ways to address domains with sparse reward signals.

One category of methods for dealing with sparse rewards seeks to enhance the agent's exploration strategy. For example, one way to motivate exploration is to look for novel states to visit. Intrinsic motivation approaches reward the agent for visiting new and unexpected environment states. Often the agent may maintain a model of the environment and state transition function. The agent is intrinsically rewarded when it encounters a sensation that disagrees with its model predictions (Hester and Stone, 2015). As the model is continually refined, these intrinsic rewards will become less frequent and the agent will have visited more of the state space. Another approach (Bellemare et al., 2016) encourages visitation of novel states by using a neural network to maintain an approximate state visitation count.

Another category of methods rely on videos of the correct solution to the task, typically obtained by recording a human or oracle performing the task. Next, the reward function that is motivating the behavior of the human or oracle can be estimated by using a technique called inverse reinforcement learning (IRL). Inverse reinforcement learning is typically quite hard because of the vast number of possible reward functions that could be correctly estimated as driving the policy of the oracle agent. However, there has been some preliminary work on IRL in the context of robotics learning (Finn et al., 2016).

Another category of methods are designed to solve the Optimal Rewards Problem (ORP). Early work on optimal rewards (Singh et al., 2010) used an evolutionary approach to create reward functions that lead to success across environments. The recent work of Liu et al (Liu et al., 2014, 2012; Sorg et al., 2010) describe a multiagent architecture where each agent employs a gradient-based algorithm to learn its own reward function. Evaluations on two multiagent domains show that the learned reward functions outperform simply using a team-based reward. However, this approach relies on the agents being able to plan in the given domain and could encounter difficulties in a model-free setting such as HFO.

Another approach is to break the target task down into a curriculum of subtasks. Narvekar et al. (Narvekar et al., 2016) presents a methods for creating a

curriculum of source tasks which can be leveraged to learn a target task. In order to create new source tasks, a parameterized model of the domain is leveraged along with observed trajectories of the agent performing the target task. Results are shown on Ms. Pac-Man and a version of Half Field Offense that uses high-level discrete actions. The results show that the use of a curriculum can speed up learning on the target task and lead to higher asymptotic performance. In contrast, our work uses a target task (Soccer) featuring a reward function that is too sparse to learn alone. The curriculum is necessary to even begin to tackle this target task.

Broadly, curriculum learning is based on the idea that skills and knowledge gained in one task may be transferred to the next task. Transfer learning has been the topic of much study with several approaches being proposed in the area of deep reinforcement learning. In particular the Actor-Mimic architecture (Parisotto et al., 2015) trains a single policy network that learns how to act in a set of distinct tasks by using the guidance of several expert teachers. Similarly, Rusu et al. (Rusu et al., 2015) introduce a Policy Distillation method for transferring policies between Deep Q-Networks and show that transfer can be achieved between tasks in the Atari domain.

The progressive neural network (Rusu et al., 2016) presents another approach for transfer learning. The progressive architecture learns a network column for each task and uses lateral connections between columns to allow transfer of information between older tasks and newer ones. The benefit of such an architecture is that it never forgets how to perform older tasks (since weights are frozen). However, the drawback is that the number of parameters in the network increases as each new task is added.

In the transfer learning setting, Guo et al (Guo et al., 2013) train a reward mapping function that can provide good initial guidance reward functions for new tasks. By leveraging good optimal rewards from previous tasks in the sequence, they can provide good initial guesses of guidance reward for new tasks. Such a framework could be useful extended in HFO in the context of learning task embeddings for new tasks.

Layered Learning (MacAlpine et al., 2015b; Whiteson and Stone, 2003;

Stone and Veloso, 2000) is a hierarchical learning paradigm that breaks a complex task into many smaller subtasks. Learning begins separately with each of the subtasks and then proceeds upwards in the hierarchy to the complex target task. Layered learning has been effective in both 2D and 3D RoboCup simulated soccer domains (MacAlpine et al., 2015b; Stone and Veloso, 2000). Other hierchical decomposition approaches (Bai et al., 2012, 2013) also build on the similar intuition of solving a difficult task by breaking it down into a hierarchy of smaller, solvable subtasks.

7.4 Approach

The approach we take is one of breaking down the primary task into many different subtasks such as moving to the ball, dribbling the ball, passing the ball, and kicking towards the goal. Each individual task is easy to learn using a simple reward function, and together these subtasks contribute to the overall skills an agent will need in order to play an effective game of soccer. By first learning each of the subtasks, the agent can harness this knowledge to then tackle the complex original task. The sparse goal reward will no longer be as daunting because the skills the agent has learned can be leveraged in order to access more promising parts of the state space than random exploration could.

In this chapter, both the subtasks and their reward functions are manually created and serve as a way of transferring domain specific knowledge to the agent. However, just the tasks alone are often not sufficient for learning, especially if one task requires skills that are learned in another. Instead, Curriculum Learning posits that if the agent is presented with a sequence or curriculum of tasks in such as way that the knowledge learned in each task is utilized in the next task, the agent will be able to more quickly and effectively learn to perform the target task. In the next section, we present the different subtasks that contribute to the soccer curriculum.

7.5 Move To Ball Task

In the move to ball task, the agent and ball are initialized randomly on the field. The agent is rewarded for approaching the ball, specifically for minimizing the distance between itself and the ball. Episodes end when the agent reaches the ball or if the ball is not touched within 100 timesteps. This is the most basic task and has no prerequisites. Additionally, it corresponds directly to the first part of the reward function given in Chapter 4.1.

$$r_t = d_{t-1}(\text{agent}, \text{ball}) - d_t(\text{agent}, \text{ball})$$

7.6 Kick to Goal Task

The kick to goal task initializes the agent randomly with possession of the ball. The agent is rewarded for minimizing the distance between the ball and the goal. Episodes end when a goal is scored, the ball goes out of bounds, or the ball is untouched for 100 timesteps, or a maximum of 500 timesteps pass. In order to correctly perform this task, the agent must already know how to approach the ball, otherwise it will be unable to kick towards the goal more than once. The reward function of the kick to goal task corresponds directly to the second part of the reward function given in Chapter 4.1.

$$r_t = d_t(\text{ball}, \text{goal}) - d_{t-1}(\text{ball}, \text{goal})$$

7.7 Soccer Task

This is the classic soccer task: the agent is initialized randomly on the offensive half of the play field, away from the ball. The agent is only rewarded one point for scoring a goal and gets zero reward for all other timesteps. In order to perform the soccer task, the agent should know how to move to the ball and kick towards the goal. Without understanding these subtasks, the sparse goal reward is very difficult

to obtain and will render the task impossible for the agent. The agent is given 500 steps to solve this task, however the task is also terminated if the ball is kicked out of bounds or remains untouched for 100 timesteps.

Having presented the move to ball and kick to goal subtasks, as well as the soccer target task, the next section presents methods that allow a single agent to learn all of these tasks.

7.8 Task Embedding

One challenge of learning from a curriculum of tasks is informing the agent which task is currently active. Because the tasks all share the same state space and action space (defined in Chapter 2.5.1), the agent does not implicitly know which task it is in. This is mainly a problem if the reward functions for the different tasks in the curriculum are very different. An unaware agent may inadvertently accrue negative rewards simply by performing the wrong actions for the current task. Since all tasks have identical state representations, it is necessary to identify which task is currently active, so the agent knows which reward function is governing the environment.

Moreover, the question of how to encode a task is crucial. An ideal task encoding would capture enough information to allow the agent to understand how the task should be performed, but otherwise remain as compact as possible. Let us assume we have a fixed curriculum with some finite set of n tasks. Each task may be represented by an integer $0 \dots n$ denoting its index in the set. However, an integer task representation, when paired with a neural network, is less than ideal. Typically, a n-dimensional one-hot representation could be used. However, such a representation provides little information beyond which task is active: it is a very simple representation of the task.

Instead of a one-hot vector, the architectures presented below use an embedding layer to create a vector representation of each task $\mathcal{T}_1 \dots \mathcal{T}_n$. These vector representations may be thought of as similar to word embeddings (Mikolov et al., 2013), where the content conveyed is the objective of the task rather than the mean-

ing of the word. In contrast to the one-hot representation, a vector representation is more descriptive and can capture similarities between tasks. Specifically, the task embedding is created by projecting the n-dimensional one-hot task representation $i \in \mathbb{R}^n$ into a d-dimensional embedding vector $\mathcal{T} \in \mathbb{R}^d$ using an embedding matrix $W^{emb} \in \mathbb{R}^{d \times n}$.

$$\mathcal{T} = W^{emb} i$$

Where the embedding weights W^{emb} are initialized randomly, but can also be learned as the network is updated. The size of the one-hot vector i is determined by the number of unique tasks in the curriculum, and the dimension d of the task embedding vector is a hyperparameter. The experiments in the following sections use 8 and 128-dimensional embedding vectors.

The following sections explore different approaches to using this task embedding vector. Specifically, the *State Embedding* and *Weight Embedding* approaches use the task embedding in different ways to inform the agent of how it should act in the current task.

7.9 State Embedding Architecture

The state embedding approach builds on the standard actor-critic architecture (presented in Chapter 4) for learning in parameterized action space. The state embedding approach concatenates the task embedding vector with the agent's state representation. This happens in states processed by both the actor and critic networks. By making the embedding a part of the state, the agent can choose to pay as much or little attention to the embedding as it desires.

The advantage of the state embedding approach is simplicity. However, this approach has the disadvantage of increasing the size of the agent's state representation, potentially making learning task more difficult. Additionally, because the task embedding is a part of the state space, we do not modify the weights of the task embedding matrix W^{emb}, meaning that the embedding vectors for each task remain static. The state embedding architecture is depicted in Figure 7.1.

7.10 Weight Embedding Architecture

The weight embedding approach conditions the activations of the actor and critic networks on the task embedding vector. Specifically, the activations of the task embedding vector multiplicatively interact with the activations of the agent's network:

$$o = W^{dec}(W^{enc}h \odot W\mathcal{T}) + b$$

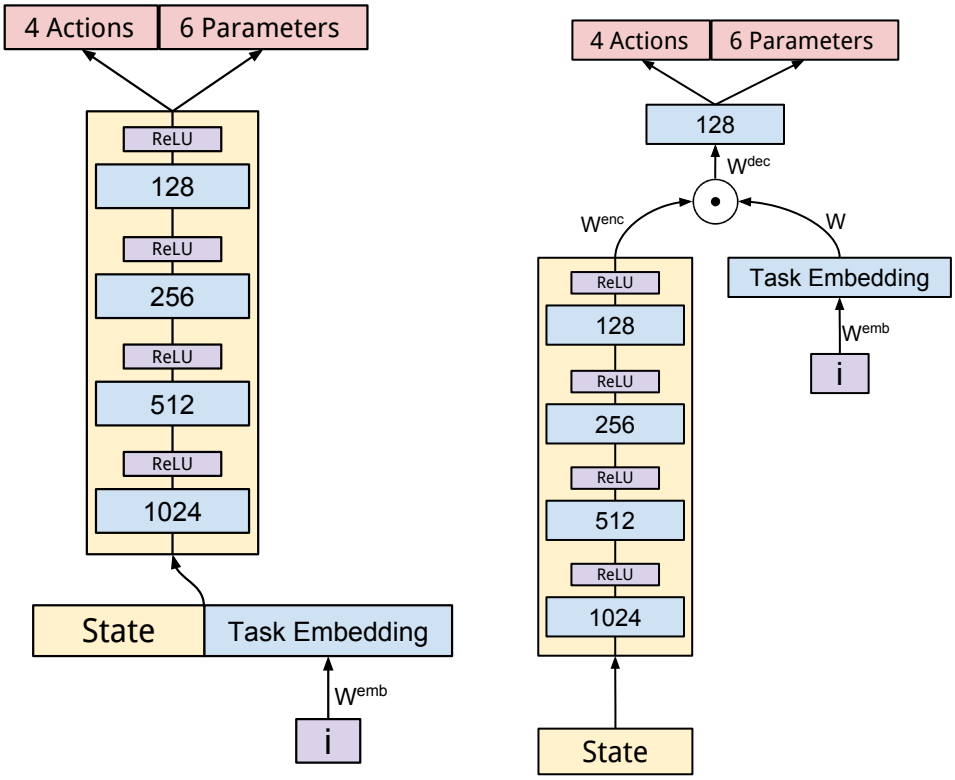
Where $h \in \mathbb{R}^d$ are the activations of layer n of agent's network, $\mathcal{T} \in \mathbb{R}^c$ is the task embedding vector. These quantities are transformed into the same f dimensional space using matrices $W^{enc} \in \mathbb{R}^{f \times d}$ and $W \in \mathbb{R}^{f \times c}$. They are then element-wise multiplied ($\odot$ operator) and transformed into the decoded output space o by weight matrix W^{dec} and bias vector b.

It is possible to vary the dimension of the embedding space f: Larger f will be able to capture more information at the cost of more parameters to optimize. In the experiments that follow we employ a 128-dimensional f, but expect that a wide variety of dimensionalities would provide similar results.

Additionally, it is also possible to decide which layer or layers of the agent's network interact with the task embedding. Intuitively, layers lower in the network have more to do with processing the state inputs, while the higher layers are more concerned with action selection and Q-Value estimation. We choose to use the 2nd-to-last layer of the actor network to interact with the task embedding vector. This way, the action selection will be highly influenced by the task embedding, but the network will still have a chance to share common lower layers across tasks.

At a high level, the *weight embedding architecture* allows the network to conditionally alter its output as a function of the active task. Additionally, in contrast to the state embedding approach, the weight embedding does not suffer the disadvantage of increasing the dimension of the state space. Finally, the weight embedding approach allows the agent to learn the weights W^{emb} of the task embedding vector, and customize it for the task at hand. The weight embedding approach is motivated by the action conditional video prediction architecture used in (Oh et

al., 2015), in which predicted video frames were condition on the action selected by the agent. Both task embedding architectures are shown in Figure 7.1.



(a) State Embedding

(b) Weight Embedding

Figure 7.1: **Task Embedding Architectures**: The current task index i is provided as input to the network and converted into a task embedding vector. The State Embedding architecture simply concatenates the task embedding vector with the current state. In contrast, the Weight Embedding architecture uses a multiplicative interaction between the embedding and the second-to-last layer of the network. Note that the same embedding operations are applied to both the actor and critic networks.

7.11 Curriculum Ordering

The previous section presented two different ways of informing the agent of the current active task. However, a higher level question is how the tasks in the curriculum should be ordered through time to achieve the best learning performance. Admittedly, the notion of "best learning performance" is ambiguous, and could be interpreted to mean a) the fewest number of episodes required to reach a target level of performance on all tasks, or b) the number of episodes required to reach a threshold level of performance only on the final task. In this chapter we strive for simultaneous competence on all tasks in the curriculum and judge the performance of agents by the number of episodes that are required to learn to perform all the tasks in the curriculum.

The next section introduces random and sequential curriculum learning methods for ordering tasks through time. Both methods are orthogonal and complimentary to the task embedding approaches presented in Section 7.8.

7.11.1 Random Curriculum

The random curriculum selects a random task from the set of possible tasks at each episode. This simple approach serves as a baseline for future comparison. The advantage of a random curriculum is that, given an infinite learning time, each task will be visited an infinite number of times, so if the agent is capable of learning all the tasks, this curriculum will permit it to do so. The disadvantage is that the agent may be presented with advanced tasks before it has developed the necessary capabilities to address them. Thus, all other things being equal, we expect that the random curriculum will require a large number of episodes in order to reach a threshold of competence across all tasks.

7.11.2 Sequential Curriculum

The sequential curriculum, shown in Algorithm 3, utilizes prior knowledge in the form of a task ordering. It begins by presenting the easiest task in the curricu-

lum again and again until the agent develops competence. The sequential curriculum then moves to the next easiest task and presents that task until mastery. This pattern continues until either the agent masters all of the tasks, or it loses competency on some previous task. In the latter case, the sequential curriculum will revisit the previous tasks long enough for competency to be regained.

By leveraging prior knowledge about the difficulty of different tasks, and learning each task in order of easiest to most difficult, we expect that the sequential curriculum will allow the agent to more quickly become compentent on the full set of tasks than the random curriculum. For example, by first learning the Move To Ball and Kick To Goal tasks, the agent can then start to tackle the more difficult Soccer task.

In order to reassess the agent's performance on prior tasks, we leverage rewards from routine policy evaluations. More specifically, every 10,000 iterations the agent's current policy is evaluated across all tasks. The resulting performance is used to decide if competence on the current task has been achieved or if a previous task needs to be revisited. Since policy evaluation is a part of the learning algorithm (Chapter 4), no additional work is needed to determine if performance is falling short on previous tasks.

To determine if an agent has mastered a task T, it is necessary to have an upper bound R_T^{max} on the possible return that is achievable in each task. This upper bound is used to determine a performance threshold for that task of $.8 * R_T^{max}$. If the agent is above the performance threshold, we say it has mastered the task and is allowed to proceed to the next text. Conversely, if it is below the threshold, it needs to spend more time learning that task.

In order to determine R_T^{max} for each task, we rely on human knowledge of the reward function combined with prior performance of agents trained on that particular task. In summary, performance thresholds are used to determine when tasks need to be revisited and must be manually defined for each task in the curriculum.

Having presented the Random and Sequential Curricula for ordering the sequence of tasks, as well as different embedding methods to make the agent aware of the current task, we now present results of curriculum learning in the various HFO

subtasks.

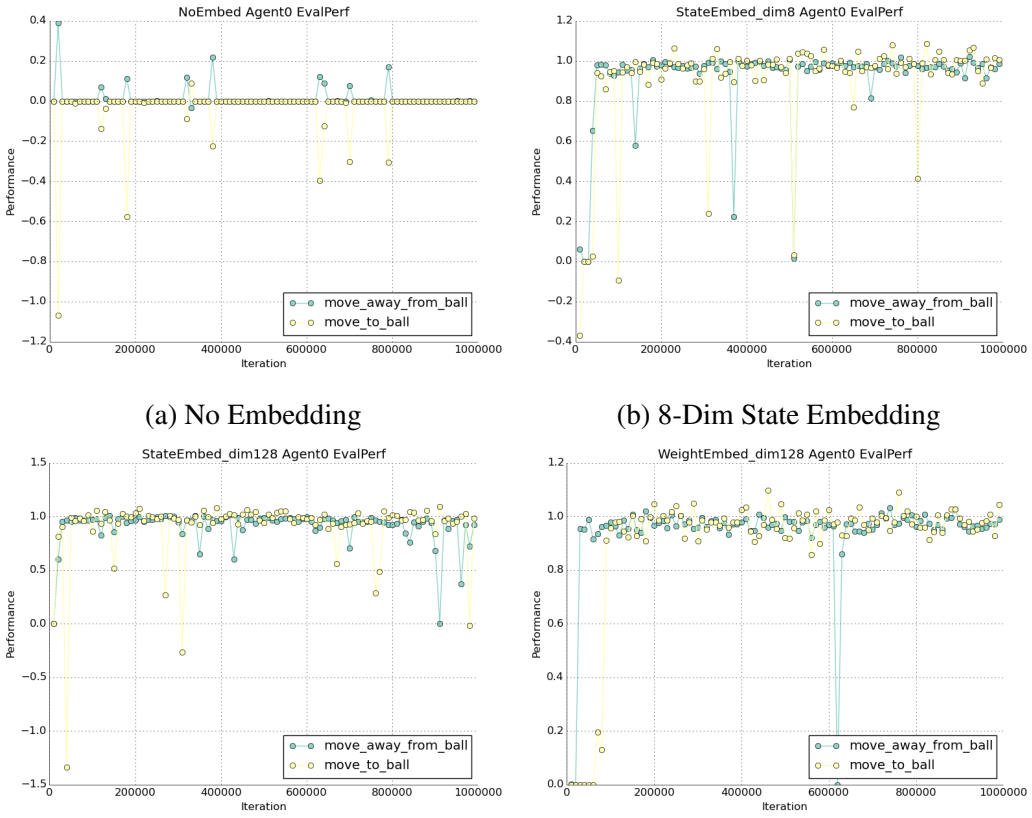
Algorithm 3 Sequential Curriculum Learning

```
1: procedure LEARN SEQUENTIAL CURRICULUM
2:     current task index i = EvaluateTasks()
3:     for iteration < max iter do
4:         PlayEpisode(T_i)                          ▷ Learn on task T_i
5:         if iteration %10,000 == 0 then
6:             i = EvaluateTasks()                   ▷ Update current task index
7:
8: procedure EVALUATETASKS
9:     for i ∈ {1 ... |T|} do
10:        average return R^avg = Evaluate(T_i)
11:        if R^avg < .8 × R_T^max then
12:            return i
       return |T|                                    ▷ Return final task
```

7.12 Task Embedding Sanity Check

To illustrate the necessity for task embeddings (Section 7.8), we present a curriculum composed two tasks: *MoveToBall* and *MoveAwayFromBall*. As their names imply, these tasks respectively reward the agent for approaching and moving away from the ball. The maximum expected returns for the move to ball and move away from ball tasks respectively are 0.6 and 0.7.

Results in Figure 7.2 show that without the benefit of a task embedding the agent has no way of knowing whether it should approach or retreat from the ball. As a result the agent cannot master either of the tasks within the one-million iterations allocated. However, when the agent is made aware of the current task using an embedding, both the state embedding and weight embedding approaches result in mastery of both tasks. All embedding approaches perform equally well on this curriculum, and simply illustrate the necessity of informing the agent about the current task. The next section presents a more challenging curriculum.



(a) No Embedding

(b) 8-Dim State Embedding

(c) 128-Dim State Embedding

(d) Weight Embedding

Figure 7.2: **Move To/Away From Ball Performance**: Evaluation performance as a function of iteration on MoveToBall and MoveAwayFromBall tasks for three different task embeddings. No embedding results in low performance on both of the tasks. Embedding the task vector as a part of the agent's state space (either using 8 or 128 dimensions) results in both tasks being learned. Finally, using the weight embedding approach, both tasks are also quickly learned. Performance (plotted on the y-axis) is the achieved return divided by the maximum possible task-specific return.

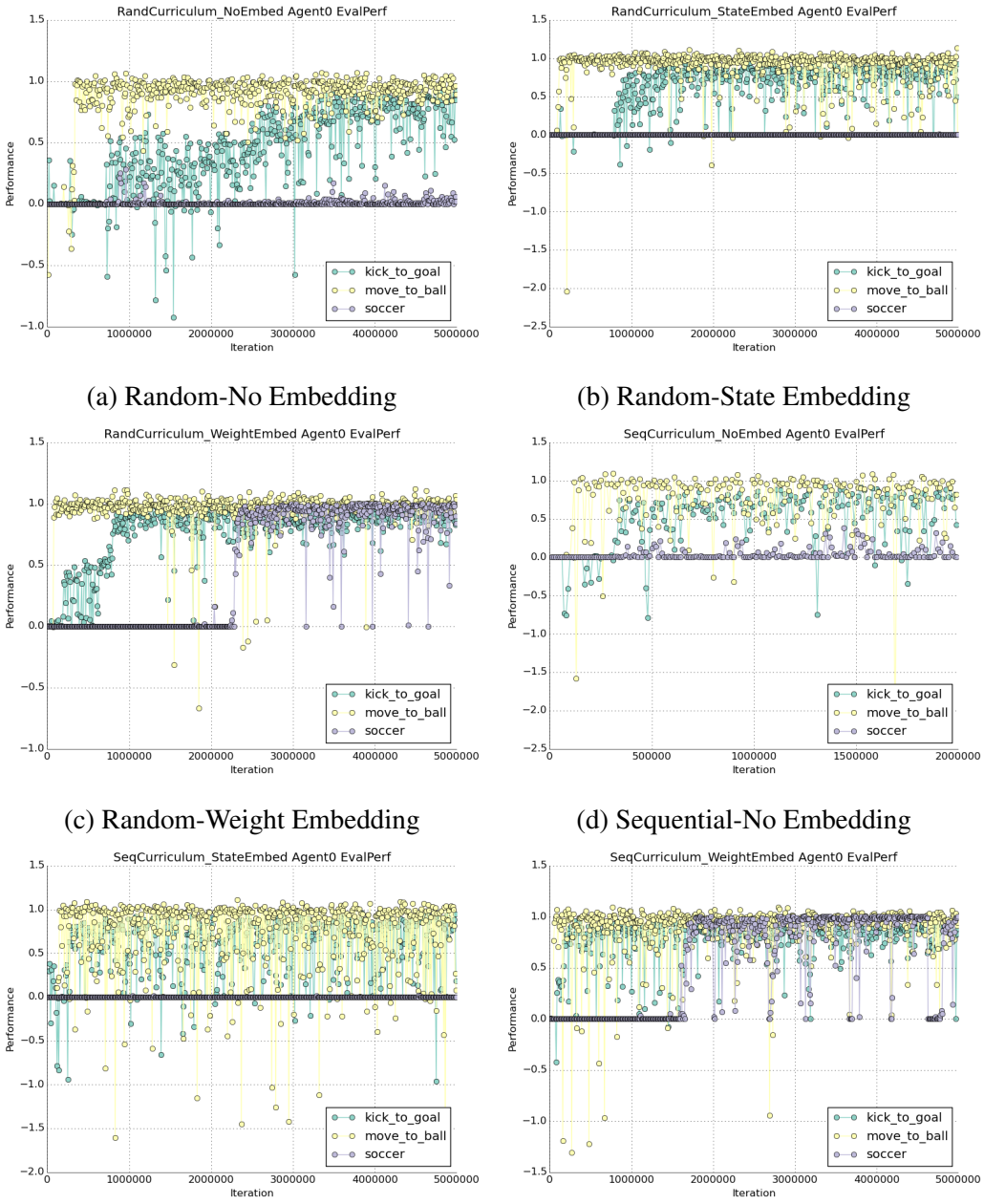
7.13 Results: Soccer Curriculum

The intention of the soccer curriculum is to learn a policy for scoring on an empty goal. This curriculum employs three tasks: *MoveToBall*, *KickToGoal*, and *Soccer*. MoveToBall and KickToGoal can be learned on their own, but both skills are required for Soccer. The maximum expected returns for MoveToBall, KickToGoal, and Soccer tasks respectively are 0.6, 0.4, and 1.0. We present results comparing a) Random Curriculum with the Sequential Curriculum and b) the three different methods of task embedding - No Embedding, State Embedding, and Weight Embedding.

Results, shown in Figure 7.3 show that both the sequential and random curriculums paired with no task embeddings or state embeddings fails to yield appreciable learning on the soccer task. The move to ball and kick to goal tasks are learned in varying amounts of time. Only the weight embedding approach results in stable learning of all three tasks.

Comparing the random and sequential curricula, we see that in all cases, the sequential curriculum results in tasks being learned in fewer episodes than the random curriculum. In the case of this 3-task sequence, the random curriculum is largely wasting time by presenting the soccer task before the other two tasks have been mastered. We suspect this effect would be more pronounced as the number of tasks in the curriculum grows and the number of dependencies between tasks increases.

Using the task embedding and curriculum in conjunction allows the agent to robustly learn the soccer task using only the sparse goal reward. However, there is a price to be paid: in particular, it takes approximately one order of magnitude more updates than the hand-designed reward function to learn the soccer task (see Chapter 4.7 for learning curves). However, we suspect that for tasks more complicated than soccer it will be impossible to hand-design a reward function and a curriculum learning approach may be the only viable alternative.



(a) Random-No Embedding

(b) Random-State Embedding

(c) Random-Weight Embedding

(d) Sequential-No Embedding

(e) Sequential-State Embedding

(f) Sequential-Weight Embedding

Figure 7.3: **Soccer Curriculum Performance**: These plots compare the performance of sequential and random curriculums using no task embeddings, state embeddings, and weight embeddings. Each plot shows the performance of the agent when evaluated for 100 episodes on each of the tasks: MoveToBall, KickToGoal, and Soccer. Sequential curriculum with weight embeddings is the quickest to reach high performance across all tasks.

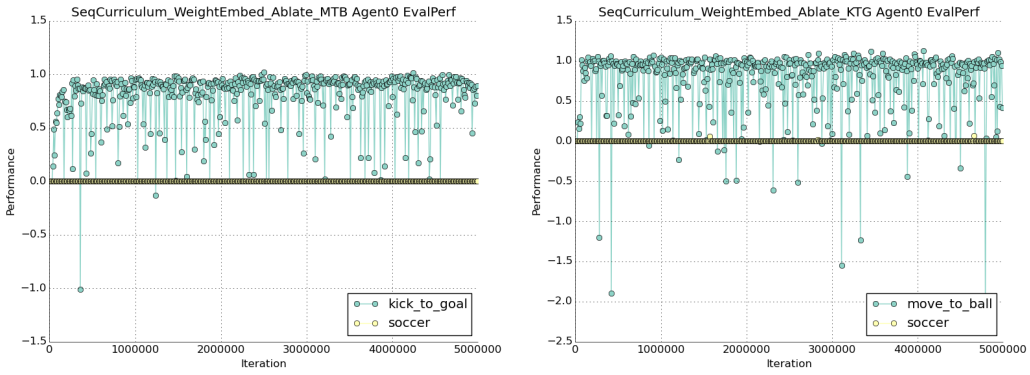
7.14 Ablation Experiment

The importance of the full set of tasks in the soccer curriculum may be further analyzed by performing an ablation experiment in which tasks are omitted from the curriculum. Specifically, we select the best performing architecture, the sequential curriculum using the weight embeddings, and analyze learning performance when either the Move to Ball or Kick to Goal task is eliminated from the curriculum.

Figure 7.4 shows the learning curves for the ablation experiments. In both cases, the first task is mastered (either Kick to Goal or Move to Ball), but no progress is made on the Soccer task. Visualizing the best-scoring policies, it is apparent that the agent who did not have the move to ball task cannot reliably approach the ball (Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/SeqCurric_Ablate_MTB.mp4`).

Interestingly, the agent without the kick to goal task could reliably move to the ball in the Move to Ball task, but did not reliably approach the ball in the Soccer task (Video: `http://www.cs.utexas.edu/~larg/hausknecht_thesis/SeqCurric_Ablate_KTG.mp4`). Thus even though the knowledge (e.g. weights) for moving to ball is present, it is not expressed in the Soccer task. The lack of reward for moving to the ball in the Soccer task likely accounts for this unexpected behavior.

The results of this ablation experiment, while perhaps unsurprising, lend credence to the notion that each task in the curriculum is effectively teaching the agent a particular skill, and if coverage of the skills required to perform the target task is incomplete, then target task performance suffers. Thus, for successful curriculum learning it is necessary to identify all the skills needed in the target task and construct subtasks to train each skill.



(a) Ablate MoveToBall (b) Ablate KickToGoal

Figure 7.4: **Ablation experiment**: In the soccer curriculum, when either the Move to Ball or Kick to Goal task is removed from the curriculum, subsequent learning fails on the Soccer task.

7.15 Analysis of Sequential Curriculum

This section analyzes the tasks visited by the sequential curriculum and demonstrates that 1) the sequential curriculum must continue to revisit earlier tasks in order to maintain high performance and 2) if the sequential curriculum does not revisit earlier tasks, performance on the target task can become unstable. Figure 7.5 shows that a stable sequential curriculum continues to visit all of the tasks in order to maintain high performance across the set. Moreover, if the sequential curriculum does not have the ability to revisit older tasks, we observe that performance on the soccer task, the final task of the curriculum, while initially high, subsequently drops. Thus, mastery of the full set of tasks comes only through continual re-exposure to every task in the curriculum.

In future work it would be interesting to maintain separate replay memory buffers for each task. Then, rather than having to revisit older tasks in the curriculum, an agent could simply update its policy using experiences saved in that task's replay buffer. Since these experiences were sufficient to learn the task in the first place, it is reasonable to assume they could be sufficient to maintain knowledge of that task without needing to sample new experiences from it.

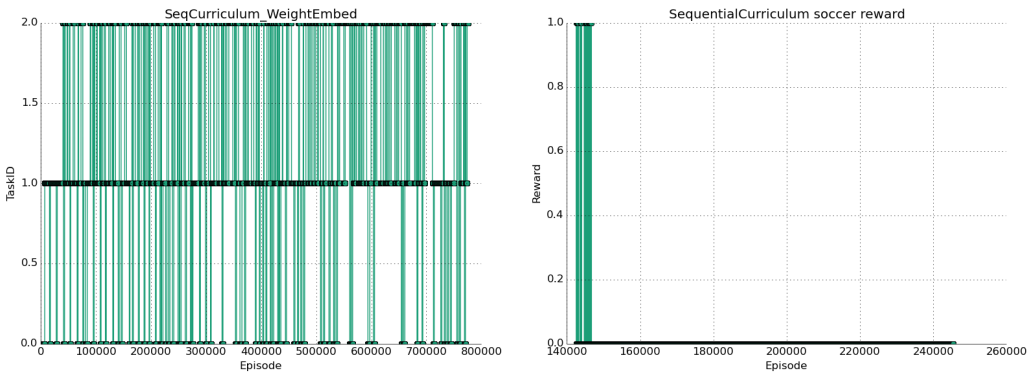


Figure 7.5: **Sequential Curriculum does not learn stable policies without the ability to revisit older tasks**: Left: to maintain high performance on all tasks, the sequential curriculum must periodically revisit all the tasks. Plotting TaskID (0 is MoveToBall, 1 is KickToGoal, and 2 is Soccer) over time shows that all tasks are periodically visited when using the sequential curriculum learning presented in Algorithm 3. Right: If revisits to older tasks are disallowed, after learning MoveToBall and KickToGoal, the performance on soccer shows initial success (goal scoring episodes terminate with reward 1) followed by unlearning and subsequent failure. To maintain a high success rate on the final task, it is necessary to revisit the MoveToBall and KickToGoal tasks.

Another possibility would be to maintain separate functional regions, or separate networks entirely, for each different task. This way, not forgetting how to perform a task would be as simple as freezing the weights for the functional region associated with that task. The weight embedding approach moves in this direction by allowing the network to tailor its activations and embeddings for each task.

7.16 Chapter Summary

This chapter presented a framework for learning from a curriculum of tasks. Curriculum learning is motivated from the difficulty at designing reward functions for complex tasks that are informative, hard to exploit, and don't lead to biased or suboptimal policies. Instead of trying to design a reward function for a difficult task, it is often possible to break that task down into smaller subtasks, where each subtask corresponds to a skill that is necessary in the difficult task. The reward functions for each subtask are much easier to design and harder to exploit. By learning each task in the curriculum, an agent works its way towards developing the necessary skills to tackle the difficult task. Additionally if the skills encoded in each subtask are fully learned, the difficult task can use a highly sparse reward function that will lead to little bias in the resulting policy.

However, tackling a sequence of tasks introduces new difficulties. Namely, the agent needs to be informed of the task that is currently active. This chapter presented two methods for converting the current task index into a vector embedding and providing this embedding to the agent. The *state embedding* method concatenated this vector with the agent's current state while the *weight embedding* method conditioned the activations of the network on the current task. We presented experimental results showing the necessity of using some type of task embedding when learning a curriculum of tasks and showed that, in general, the weight embedding approach offered better learning performance compared to the state embeddings.

Finally, we demonstrated that the order in which the tasks are presented to the agent can have an impact on the number of episodes required to learn. We introduced a sequential curriculum which presents tasks sequentially starting from

the easiest and working towards the hardest. Compared to randomly selecting a task at each episode, the sequential curriculum reduces the time needed to master the full curriculum of tasks. Results were shown for a curriculum of tasks that lead to a policy capable of approaching the ball and scoring goals.

In general, curriculum learning offers an appealing way of avoiding hand-designed reward functions. However, it requires a substantial amount of expert knowledge in the form of a task decomposition, reward functions for each of the subtasks, and an ordering over difficulty of tasks. Additionally, results show that stable learning requires revisits to each of the subtasks and much more total experience than a hand-designed reward function. In light of these difficulties, it makes sense to hand-design reward functions whenever possible, and reserve techniques like curriculum learning for tasks too complex to design informative reward functions.

This chapter concludes the contributions of this dissertation. The final chapter summarizes and concludes.

Chapter 8
Related Work

Although there has been much research in the field, this is among the first dissertations about deep reinforcement learning. Notable exceptions include Reinforcement Learning for Robots Using Neural Networks (Lin, 1992) and Motor Skill Learning with Local Trajectory Methods (Levine, 2014). Compared to the related work sections found in most chapters, this chapter presents a more high-level perspective on the field of deep reinforcement learning and attempts to elucidate the relationship between the work in this thesis and the larger body of deep reinforcement learning. This chapter may be understood at a high level on its own.

Due to the scalability issues of tabular reinforcement learning, it is common to employ some type of function approximation when learning a value function. The idea of using neural networks as function approximators for reinforcement learning agents has existed for quite some time. However, neural networks consisting of more than a few layers were notoriously difficult to train using backpropagation, suffering from vanishing gradients, which today is recognized as a problem stemming from the ubiquitous use of the sigmoid nonlinearity which provides informative gradients only over a small portion of the activation range.

Instead of neural networks, many reinforcement learning practitioners chose to use linear function approximation (LFA), which yielded more traction for theoretical analysis of convergence properties (Tsitsiklis and Roy, 1997; Melo and Ribeiro, 2007; Melo et al., 2008). The downside of LFA was the lack of representational power: in practice it was necessary to either engineer good features for the agent to use or stick with simple domains. Other researchers preferred more powerful function approximators such as forests of decision trees (Hester and Stone, 2015). These models sacrificed the theoretical guarantees of LFA but often worked better in practice.

Despite the difficulties of backpropagation, some researchers persisted with neural network function approximators. However, they used evolutionary approaches

to optimize either the topology or parameters of the networks, sometimes in combination with backpropagation (Whiteson, 2007), and sometimes alone (Stanley and Miikkulainen, 2002). While ineffective at optimizing large networks, evolutionary methods could often discover compact networks that did a reasonable job at representing the value function or policy (Hausknecht et al., 2013).

The neural network resurgence began with Deep Belief Networks showing impressive generative modeling capabilities (Hinton et al., 2006). More results soon hinted at the ability of unsupervised neural networks like auto-encoders to discover features and learn representations (Le et al., 2012). Soon after, discriminative neural networks with many stacked layers started to be vastly more tractable to train. This was made possible by the abundance of parallel computing power embodied by GPUs as well as the use of nonlinearities like Rectified Linear Units (ReLU) which went a long way to ameliorate the problem of vanishing gradients. Neural network ideas developed twenty or thirty years ago such as convolutional networks (LeCun et al., 1998) and Long Short Term Memory (Hochreiter and Schmidhuber, 1997) were suddenly not only applicable, but yielding state-of-the-art results on nearly every domain they were applied to. Deep discriminative neural networks were performing tasks like image recognition that were considered impossible just a couple years before.

The power of neural networks was evident, and the time was right for them to be applied to reinforcement learning. Some early works began to show that like image recognition, reinforcement learning agents could now harness pixels representations (Lange et al., 2012). However, stability of deep neural networks used for reinforcement was less than desirable. The breakthrough came in the form of the Deep Q-Network (Mnih et al., 2015), which showed robust learning on a variety of Atari games using raw pixels as input. From a technical standpoint, DQN demonstrated the use of target networks, experience replay, and adaptive learning rate optimizers to stabilize neural network training in the context of reinforcement learning. With these developments, deep reinforcement learning was born and with it came a variety of works extending and improving DQN. Like with the neural network resurgence, old ideas from reinforcement learning were suddenly not only appli-

cable, but also offering state-of-the-art performance when paired with deep neural networks. Some ideas that found traction were double Q-Learning (van Hasselt et al., 2015), advantage learning (Wang et al., 2015; Schulman et al., 2015b), prioritizing experience replay (Schaul et al., 2015), and recurrent networks (Hausknecht and Stone, 2015).

Extending deep reinforcement algorithms into continuous action spaces is an important step towards real-world domains such as robotics, which require agents to use continuous controls like real-valued torques applied to actuators. Several approaches for continuous control using deep neural networks have emerged. Actor-critic architectures such as Deep Deterministic Policy Gradients (Lillicrap et al., 2015) (see Section 2.7) maintain separate neural networks for learning a policy (actor) and a value function (critic). However, the learning is still driven by traditional Bellman-style updates. Alternative architectures such Normalized Advantage Functions (NAF) represent the Q-function in such a way that its maximum can be determined easily during the Q-learning update. Parameterizing the Q-function as a quadratic allows NAF to use only a single network, making it considerably simpler than DDPG (Gu et al., 2016). Other approaches to continuous control avoid Bellman updates altogether, instead preferring policy gradients. Policy gradient methods (Schulman et al., 2015b,a) directly alter the policy of the agent in the direction of higher rewards, simply by increasing the probabilities of actions in good trajectories. All of these methods show promise in continuous action space and more work is needed to understand the tradeoffs between the different algorithms. Since robots typically have tens to hundreds of actuators that all need to be controlled at the same time, future algorithms will need to be able to function in higher-dimensional action spaces than are possible currently.

Novel advances in deep neural networks and deep reinforcement learning continue to drive the field. One of the most promising recent directions is external memory. Integrating read/write memory into a deep reinforcement learning agent can help the agent better recall events or sensory input from many steps ago. This information can be helpful for dealing with partial observability and creating history-aware policies. The LSTM controller in models like DRQN takes a first

step in this direction, but differentiable memory agents are capable of handing dependencies far more complex than LSTM alone (Oh et al., 2016). However, the effective size of the external memory is still limited and addressing mechanisms need to improve before external memories can grow hold and recall hundreds or thousands of items.

There are several notable bastions that even today remain unconquered. One of the most prominent is the lack of high performing model-based reinforcement learning approaches. The typical promise of model-based reinforcement learning is the ability to learn good policies from fewer experiences by needing to revisit states less often. In the Atari domain, high quality predictions of next states have already been achieved (Oh et al., 2015). However, when planning over long horizons, small errors in models tend to accumulate and magnify, leading model-based deep RL agents to make incorrect decisions. These errors are often cited as the primary reason why model-based RL has yet to match the power of model-free approaches like DQN. For this reason, Stochastic Value Gradients (SVG), one of the few existing approaches for model-based RL (Heess et al., 2015) computes value gradients using historical trajectories rather than ones simulated by the model. To address this challenge would require more precision on the part of deep neural networks or a better way of estimating model error and an understanding of when to stop planning.

As reinforcement learning agents become more capable, they begin to be exposed to domains that are increasingly complex. As a by-product, reinforcement learning practitioners are beginning to realize that the design of reward functions for complex tasks is no longer scalable. Hand-designed reward functions are subject to being gamed by agents that can find a way to generate more reward without performing the task as intended (Amodei et al., 2016). Additionally, current reinforcement learning agents have trouble with highly sparse rewards and often work best with reasonably frequent rewards. In general, designing non-sparse rewards is harder than the design of sparse rewards, which can often consist of simply giving positive reward at a goal state and zero reward otherwise. There are several possible directions for addressing this challenge.

First, better exploration through reward shaping (Ng et al., 1999) or intrin-

sic motivation (Chentanez et al., 2005) could hold the key to learning with sparse rewards. There has been work on improving exploration in deep reinforcement learning agents using a variety of methods (Stadie et al., 2015; Bellemare et al., 2016). Future work on improved exploration would fit well with model-based reinforcement learning since learning a model of the environment has no dependency on receiving frequent rewards. Thus, a model-based agent may be particularly suited for a domain with highly sparse rewards.

Other approaches to learning with sparse rewards seek to leverage external sources of information. Inverse reinforcement learning (IRL) assumes access to only expert trajectories and will attempt to estimate a reward function that explains the behavior of the experts. Work in deep-IRL shows promise, but is still very young (Wulfmeier et al., 2015). Imitation learning and learning from demonstration also fall into the category of methods that rely on external information rather than rewards, but have difficulty with policy improvement in the context of deep RL: it is possible to learn a policy by mimicking expert data, but trying to improve that policy is far more difficult.

Finally, transfer and curriculum learning seek to leverage skills from related tasks towards a new task, perhaps one featuring sparse rewards. There has been a reasonable amount of work on transfer learning for deep RL (Rusu et al., 2016; Parisotto et al., 2015; Rusu et al., 2015), with findings generally suggesting that positive transfer is quite possible in this context. Curriculum learning in Chapter 7 echoes this trend. We are beginning to struggle with the limitations of scalar reward functions, and in the future, learning agents will need to be able to shape their policies in response to feedback from a diverse set of sources - other agents and humans to name a few.

Scaling deep reinforcement learning beyond simulated domains is an ongoing endeavor. In particular, there is much potential for physically embodied learning agents, e.g. robotics. The typical challenges for learning agents in the physical world are myriad: agents must handle highly noisy perceptions, work with continuous actuators that often alter behavior after continuous operation, contend with realistic factors such as hardware malfunction and limited power supply. For all of

these reasons, the majority of deep reinforcement learning research, including this thesis, has taken place in simulation. Deep reinforcement learning algorithms typically require large amounts of experience to learn good policies and the prospect of collecting hundreds of thousands of episodes of robot experience is daunting. However, there are a few researchers who have made notable headway learning on physical robots (Levine et al., 2015; Finn et al., 2015; Finn and Levine, 2016), employing trajectory optimization to reduce the amount of data that needs to be collect on the robot or many robots that can collect data in parallel. Further advances in robotics will likely be made through more sample efficient deep RL algorithms as well as more reliable hardware.

Advantages in single agent capabilities will have to also come with advances in interpretability and safety of agents. This is particularly important when agents begin to inhabit or interact with real world systems. The challenges in AI safety are very real and will need to be thought about carefully to prevent learning agents from interacting harmfully with the existing world.

Future advances in deep reinforcement learning will likely leverage further advances in function approximation power of deep neural networks, or continue the trend of using using human brain regions as loose inspiration for agent based neural architectures. In addition to memory, agents stand to gain from incorporating active planning, language, reasoning, and perhaps even emotion. The work of improving the capabilities of learning agents will likely continue for the foreseeable future.

Chapter 9
Future Work, Discussion, and Conclusion

Cooperation is the process where groups of organisms act together for mutual benefit. Symbiotic relationships in nature show that cooperation is achievable and beneficial for different groups of organisms. For example, honey bees help flowers reproduce by collecting pollen which they consume for its protein content. Bacteria, living in the human digestive tract benefit humans by fermenting dietary fiber into fatty acids, which are absorbed more easily than the unfermented fiber would be. Many more symbiotic relationships exists and illustrate that by working together, independent agents can achieve far more than would be possible alone.

Cooperation and symbiosis in nature evolve over hundreds or thousands of years. However, in human society, cooperation can happen much more quickly and is often facilitated through active communication, an understanding of another person's goals, and the ability to understand how individual action can be applied to achieve a common goal. This thesis takes a small step towards better understanding the principles and architectures that aid cooperative deep multiagent reinforcement learning. We have demonstrated the approaches presented can help learning agents cooperate on shorter timescales than in nature, by leveraging techniques such as communication and sharing.

This chapter summarizes the thesis, revisits the thesis question and contributions, identifies avenues for future research, and concludes.

9.1 Thesis Summary

A main idea in this thesis is leveraging the power of deep neural networks to improve and capabilities of reinforcement learning agents. Chapter 2 reviews the fundamentals of deep neural networks and reinforcement learning. Additionally, this chapter introduces the Arcade Learning Environment and Half-Field-Offense domains which are used in later chapters. Finally, it reviews well known algo-

rithms that combine deep neural networks with reinforcement learning: Deep Q-Networks and Deep Deterministic Policy Gradients. These method represent the starting points for the algorithmic contributions of later chapters and the foundations of our deep reinforcement learning agents.

Building on the success of DQN, Chapter 3 introduces DRQN, a recurrent convolutional network intended to combat the partial observability in Atari games. Compared to standard DQN, DRQN is capable of processing information over longer time scales, conditioning the agent's policy on a greater amount of history. The ability to incorporate information through time is particularly useful when individual game screens may be obscured or occluded, in which case the locations and velocities of on-screen objects can only be successfully inferred from a history of past screens. Results demonstrate that DRQN achieves superior performance on certain Atari games that feature flickering screens. Analysis shows that the recurrent layers in the neural network are capable of inferring velocity of on-screen objects even though only a single screen is observed at a time. This result indicates that recurrent networks can be used in the context of reinforcement learning to extract salient information through time and learn a policy robust to partial observations.

Chapter 4 focuses on single agent deep reinforcement learning in parameterized-continuous actions space. To handle continuous action spaces, an actor-critic architecture based on the Deep Deterministic Policy Gradients (DDPG) algorithm is used: the actor is responsible for learning a policy whose outputs correspond to the continuous actions in the domain, while the critic learns an action-value (Q-value) estimator. Learning in parameterized action space rather than purely continuous space requires several adaptations: first we presented a method for stably bounding the activations of continuous parameters into fixed ranges. This step is necessary due to the limitations imposed by the Half-Field-Offense domain on the ranges of continuous actions. Additionally, we detailed an algorithm to combine off-policy bootstrap targets with on-policy Monte-Carlo targets when updating the critic network. We empirically demonstrate that using this hybrid update increases learning speed and policy stability. Building on these innovations, the chapter presented successful learning on single agent simulated soccer against an empty goal and

against a keeper. The architecture and updates for single agent learning are leveraged throughout the remainder of the thesis.

Chapter 5 examines multiagent scenarios in Half Field Offense. In order to coordinate multiple agents, we evaluate several approaches motivated by the idea of sharing between agents. The first approach is sharing parameters between layers of the actor and critic networks that represent each agent. Another approach is maintaining a shared replay queue that both agents store experience to and update from. Results show that both approaches can help all offense agents learn to perform the task. In contrast, without sharing parameters or memories, only one agent learns to perform the task and the other does not. Fundamentally, the sharing approaches encourage policy similarity between the agents. In domains where policy similarity engenders cooperation, sharing either parameters or experiences can lead to cooperative behavior. The limitation of this approach are seen on more challenging tasks such as soccer against a keeper, in which similarity between agents is not sufficient to guarantee success. Such complex tasks likely require more specialization of each agent.

Chapter 6 investigates different ways in which active communication between agents can enhance cooperation. We augment the actor-critic architecture with additional continuous actions that are used to broadcast real-valued messages between agents. Using this communication channel, agents have the potential to strategize and overcome deficiencies in any single agent's observations. However, as our results show, communication actions alone are often not enough to achieve cooperation. Instead algorithmic change to the learning process of both agents is needed to ensure stable communication protocols are established between the agents. One such algorithm is referred to as teammate communication gradients. This approach involves sharing gradients of communication actions between the two agents. In this manner, each agent can influence the content of the messages sent by its teammate. This approach is shown to be highly effective in domains where the reward signal is directly tied to content of the transmitted messages. Another approach, the Grounded Semantic Network, learns a communication protocol grounded in the states and rewards of the task. Our results indicate that this ap-

proach is more effective for complex tasks in which communication is used as a means to achieve some objective in the environment, rather than an end in and of itself. The Grounded Semantic Network is validated in a task in which a sighted agent must guide its blind teammate to the soccer ball using only communication. Analysis of the learned communication protocol reveals that the blind agent's policy is highly dependent on the messages sent by its sighted teammate. The limitations of the Grounded Semantic Network are demonstrated in a domain where reward is tied directly to the content of the transmitted messages. In this case, other approaches that directly optimize content of messages achieve superior performance.

Finally, Chapter 7 examines the problem of curriculum learning in the context of deep reinforcement learning agents. This chapter is motivated by the difficulty of designing effective reward functions for complex tasks. A promising alternative is curriculum learning, in which a complex task is decomposed into multiple simpler tasks, each of which corresponds to a single skill. The agent then learns to perform each task, leveraging the skills from earlier tasks to assist its learning of later tasks. We present an architecture for conditioning the weights of the agent's actor and critic networks on the current task and demonstrate successful curriculum learning across a set of three tasks that involve moving to the ball, kicking the ball towards the goal, and scoring a goal. Compared to alternative approaches, our architecture shows quicker and more stable learning across a curriculum of tasks. Through ablation analysis we demonstrate that each of the tasks in the curriculum is necessary and the skills learned by the agent are all leveraged in the final soccer task. The limitation of curriculum learning is the requirement to manually create a set of tasks as well as the increased amount of time required to learn each task in the sequence. Despite these limitations, this chapter highlights curriculum learning as a potential method to avoid the need to hand-design reward functions for complex tasks.

Taken together, these chapters illustrate the power of combining deep neural networks with reinforcement learning. Resulting algorithms are able to learn from low-level representations such as pixels, incorporate information over many timesteps, act in continuous spaces, and cooperate with other agents. This thesis has

introduced deep reinforcement learning algorithms applicable to single and multi-agent domains that take steps to accomplish these objectives. The resulting agents have only been tested in small scenarios are not ready to tackle the full ten versus ten RoboCup competition. The specific contributions are outlined in the next section.

9.2 Contributions

This thesis addresses the following questions: **1)** How can the power of Deep Neural Networks be leveraged to extend Reinforcement Learning towards domains featuring partial observability, continuous parameterized action spaces, and sparse rewards? **2)** How can multiple Deep Reinforcement Learning agents learn to cooperate in a multiagent setting?

We address each part of the thesis question separately. First, *How can the power of Deep Neural Networks be leveraged to extend Reinforcement Learning towards domains featuring partial observability, continuous parameterized action spaces, and sparse rewards?* Chapters 3-4 are devoted to answering this question. Specifically, Contribution 1 is an exploration of recurrency as a method of dealing with partial observability. The Deep Recurrent Q-Network in Chapter 3 described an algorithm for using recurrent neural network architectures to learn policies that are capable of incorporating observations across many timesteps. Such policies are more resistant to partial observations or noise in the observation space at any single timestep. DRQN forms the basis of Contribution 1.

Thesis Contribution 3 addresses the second part of this question by combining deep neural networks and reinforcement learning on domains featuring continuous parameterized action spaces. Chapter 4 forms the basis of this contribution by describing a deep reinforcement learning algorithm tailored for the parameterized continuous action space of Half Field Offense. Our algorithm has several improvements over standard DDPG, an existing deep reinforcement learning algorithm for continuous action space learning. In particular, we describe a method to ensure the agent respects the boundaries each continuous action and a modified update which

involves mixing on-policy and off-policy targets.

Returning to the question of domains featuring partial observability, Chapter 6 presents the Grounded Semantic Network, a method for learning a task-dependent communication protocol. Specifically, the GSN is able to handle the partial observability present in a multiagent setting by learning a communication protocol that transmits information to the teammate that may be lacking from their state observations. The Blind Move to Ball task (found in Chapter 6.5) is a good example of how the GSN can learn a communication protocol that combats the lack of information experienced in a partially observable domain. Formally, the GSN is part of Contribution 4: an exploration of multiagent Deep reinforcement learning. However, we view it as also contributing a perspective on overcoming partial observability.

Finally, Contribution 5 of the thesis explores curriculum learning, a method for handling tasks with sparse rewards. In fulfillment of this contribution, Chapter 7 presents a novel approach to curriculum learning that allows the policies of the agent to be conditioned on the current task. In a sense, this contribution is orthogonal to the other chapters of the thesis as the approach of conditioning the agent's policy on the task can be applied to any of the tasks examined throughout this thesis. The main difficulty is breaking a target task into smaller subtasks and the providing the extra training time required to solve each of these subtasks and then address the target task. However, the extra time investment is worthwhile in domains that are too complicated to design unbiased non-sparse reward functions.

The second part of the thesis question asks: *How can multiple Deep Reinforcement Learning agents learn to cooperate in a multiagent setting?* Thesis Contribution 4 addresses this question, by extending the deep reinforcement learning agents presented in Chapter 4 into cooperative multiagent domains and identifying several approaches that help encourage both agents to cooperatively solve multiagent tasks. Chapter 5 describes parameter sharing and memory sharing approaches which encourage policy similarity between the agents. In many tasks, similar policies can be quite beneficial, and our results show that memory sharing and parameter sharing can help both agents to learn to cooperate on a goal-scoring task that is otherwise dominated by a single agent.

To further aid coordination, Chapter 6 investigates how active communication between agents can be learned in cooperative settings. As results show, simply adding communication actions is often not sufficient to achieve a stable or fruitful communication protocol between agents. One promising method is Teammate Communication Gradients 6.2 which transfers gradients on communication actions between teammates. We demonstrate this approach is effective for a class of domains in which the content of the message is directly tied to the reward function. Another approach is the Grounded Semantic Network 6.3 which we show to be the effective in domains featuring partial observability and asymmetric information. In such domains, the GSN allows the agents to learn a stable communication protocol and successfully cooperate to solve the task. Overall, these results show that communication can be integrated with existing deep reinforcement learning methods and can also be a strongly positive force for promoting cooperation between agents.

Together, these chapters represent an exploration of deep multiagent reinforcement learning in domains featuring partial observability, parameterized continuous action spaces, and cooperative multiagent settings. We believe that they advance the power of combining deep neural networks and reinforcement learning for scaling agents towards cooperative behaviors in complex domains.

9.3 Short Term Future Work

The work presented in this thesis is only a first step rather than a complete answer to the thesis questions. There are many avenues for future work. We discuss some of short term ideas in this section and long term ideas in the next.

The work of combining advances in deep neural networks with reinforcement learning continues with more vigor than ever before. There is continued progress leveraging new neural network architectures such as differentiable read/write memory and attention to solve ever increasingly difficult tasks (Oh et al., 2016). Because there are many potential ways to improve single agent deep reinforcement learning and this thesis is specialized in multiagent reinforcement learning, we choose to focus our suggestions for future work on multiagent problems.

9.3.1 Alternative DRQN Architectures

While DRQN (in Chapter 3) showed increased performance on certain Atari games, there were others in which the performance was lower than DQN's. Comparing DRQN to DQN, one of the most striking differences is that DQN is able to utilize its convolutional layers to extract velocities of on-screen objects. In contrast DRQN's convolutional layers are given only a single screen per step as input and thus cannot recognize velocity. Instead, DRQN needs to allocate capacity of its recurrent fully-connected layers to recognize object velocity. Lower performance of DRQN in certain games may be an effect of the fully-connected capacity being used to detect object velocity rather than learning a good policy. A middle ground between both approaches would be to present DRQN with a stack of game screens at each timestep. This would allow DRQN to both extract velocity with the convolutional network as well as using recurrency to find policies resistant to partial observations. Such an approach would necessarily involve training networks with even more parameters.

9.3.2 Better Task Performance

There is much room for improving performance on Atari games and Half-Field-Offense tasks. In particular, of the HFO domains introduced in this thesis, the task of two offensive agents cooperating against a goal keeper remains unsolved. Additionally, there are a whole range of more complex HFO tasks involving more offensive teammates and more defensive opponents. Furthermore, we have not investigated the capabilities of learning agents to play defense in HFO. The performance of existing algorithms on these tasks could be established relatively quickly. Achieving higher performance on these more complex tasks would likely take more time.

9.3.3 Combining GSN and Curriculum Learning

A GSN learned over a curriculum of tasks would need to establish a communication protocol able to encompass all the necessary concepts for each task in

the curriculum. It may be possible to learn such a protocol by conditioning the activations of the GSN on the embedding of the current task, analogous to the way the agent's actor and critic networks were conditioned in Chapter 7. This would not necessarily create a single protocol, but would instead allow the GSN to switch between protocols learned for each of the tasks.

9.4 Long Term Future Work

Having discussed short term future work, we now devote some time to long term ideas for future work.

9.4.1 Teammate Modeling

In the multiagent domains throughout this thesis, each agent perceived its teammate through a limited number of features in its observation space. No explicit modeling of the teammate's behavior or intentions was done. An interesting question would be how to better model a teammate and whether or not a teammate model would be helpful in completing a task. We delve further into these questions below:

One interesting question is that of *Embodied Imitation Learning*. Specifically, consider the problem of an expert agent attempting to demonstrate how to perform a task to a novice agent. If the novice agent had direct access to the expert's states and actions, it would be straight-forward for the novice to learn a high-quality policy through supervised learning from the expert's actions. Such approaches have been heavily studied under the name of imitation learning and learning from demonstration (Argall et al., 2009; Atkeson and Schaal, 1997).

However, in the real world, a novice agent is instead embodied in the environment and can only observe the expert through it's observation space. In particular, it does not have direct access to the states seen by the expert or the actions taken. How can the novice agent relate its observations of the expert to how it should act? More specifically, how can the novice agent realize that its observation of the expert agent corresponds to an individual similar to itself that is demonstrating a solution

to the task? Furthermore, how can the novice agent infer the states seen and actions taken by the expert only from observing the effects on the state of the world? Perhaps these questions can be answered in part through a model of the expert agent. In biology, the concept of mirror neurons plays a similar role: mirror neurons allows us to recognize the similarity between ourselves and others. An analogue in a learning agent could provide a starting point for addressing these questions.

Beyond imitation learning, teammate modeling can be generally useful for cooperation (Stone et al., 2000; Barrett et al., 2012; Zhou and Shen, 2011; Kok and Vlassis, 2002). For example, it may be possible to better cooperate with a teammate when you can predict what actions it will take. Barrett showed that teammate models could be effectively utilized in a predator prey domain to achieve better cooperative performance (Barrett, 2014). However, current approaches for model-based reinforcement learning are struggling due to problems of compounding model error through time. An erroneous or mis-specified teammate model could serve to magnify these errors. Thus, we expect that advances in single agent model-based deep reinforcement learning will be necessary before teammate modeling will bear fruit.

9.4.2 Adversarial Multiagent Settings

The work in this thesis only touched on cooperative domains. A large class of multiagent problems not addressed by this thesis are adversarial multiagent domains, in which agents must compete to maximize rewards. The approaches for sharing parameters and learning communication protocols would not be ideal for adversarial situations. New architectures and algorithms would need to be developed to solve questions such as the following:

How to protect communications intended for teammates from being interpreted and used by opponents? For example, many RoboCup teams use encrypted communications in order to prevent opponents from comprehending the intent of their players through the broadcast messages. What types of deep reinforcement learning algorithms would be necessary to help encourage safe communications in the presence of an adversary? Recent work shows that neural networks can learn a system for symmetric encryption and decryption that fools an adversary network

(Abadi and Andersen, 2016).

Do deep reinforcement learning agents need to be modified in order to achieve continually improving policies in an adversarial self-play context? In particular, it would be ideal if both agents could learn approximately optimal adversarial policies through self-play. However, it is possible that one agent may dominate the other, resulting in no further learning. We suspect there are algorithms that could provide better empirical results on bootstrapping high performing and stable deep reinforcement learning policies through adversarial self-play. Similar ideas in deep learning have been examined in the context of two-player mini-max games found in Generative Adversarial Networks (GANs) (Goodfellow et al., 2014).

9.4.3 Quantitative Analysis of Reward Functions

This thesis qualitatively analyzed the reward functions of different tasks to determine whether or not they encouraged cooperation between both agents or encouraged a single agent to dominate the task. More generally, it would be interesting to see what types of quantitative analyses could be applied to reward functions. Intuitively, it seems that particular types of multiagent tasks are better solved by different approaches (e.g. sharing vs. communication), and one of the most important factors determining which approach is more effective would be understanding the properties of the reward function. There are existing algorithms such as Nash-Q and FF-Q (Hu and Wellman, 2003; Littman, 2001) that are designed for learning Nash Equilibria and correlated equilibria (Greenwald and Hall, 2003). It remains future work to use existing multiagent learning concepts to find the optimal and expected behavior of agents under different types of reward functions. Such analysis may provide insight into the successful development of multiagent reward functions or approaches for fostering cooperation between agents.

9.5 Conclusion

The fields of artificial intelligence and reinforcement learning are rapidly changing with the advent of increasing processing power and the development of

powerful general-purpose neural network function approximators. This thesis presented several algorithms that combine deep learning with reinforcement learning in single and multiagent domains. Results across multiple domains indicate that successful cooperation can be achieved by sharing parameters, replay memories, and communication protocols. We expect that further improvements in cooperation may be achieved through better modeling of teammates and improving the individual capabilities of deep reinforcement learning agents. This work represents a small step towards the development of capable and cooperative agents. Although many open questions remain to be answered, we believe there is good reason to be optimistic about what the future of intelligent agents holds.

Appendix A
Abbreviations

Abbreviation	Explanation
MDP	Markov Decision Process (Chapter 2.1)
RL	Reinforcement Learning (Chapter 2.2)
HFO	Half Field Offense Domain (Chapter 2.5)
ALE	Arcade Learning Environment (Chapter 2.4)
CNN	Convolutional Neural Network (Chapter 2.3.1)
DQN	Deep Q-Network (Chapter 2.6)
DDPG	Deep Deterministic Policy Gradients (Chapter 2.7)
DRQN	Deep Recurrent Q-Network (Chapter 3)
POMDP	Partially Observable Markov Decision Process (Chapter 3.1)
LSTM	Long Short Term Memory (Hochreiter and Schmidhuber, 1997)
GSN	Grounded Semantic Network (Section 6.3)
ReLU	Rectified Linear Unit (nonlinearity)

Appendix B
Online Materials

A directory containing the videos referenced throughout this thesis can be found at: `http://www.cs.utexas.edu/~larg/hausknecht_thesis`.

Source code for the Half Field Offense domain (Section 2.5) can be found at: `https://github.com/LARG/HFO`.

Source code for DRQN (Chapter 3) can be found at: `https://github.com/mhauskn/dqn/tree/recurrent`.

Source code for learning in parameterized action space (Chapter 4) as well as multiagent learning approaches (Chapter 5) can be found at: `https://github.com/mhauskn/dqn-hfo`.

Source code for communication and curriculum learning (Chapters 6-7) can be found at: `https://github.com/mhauskn/dqn-hfo/tree/scenario`.

Source code is provided as is without warranty of any kind. The authors stress that code above is used for research and is not of production quality. Use of the above source code will likely require time to understand, use, and extend.

References

Martn Abadi and David G. Andersen. Learning to protect communications with adversarial neural cryptography. *arXiv preprint*, 2016.

Hidehisa Akiyama. Agent2d base code, 2010.

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *CoRR*, abs/1606.06565, 2016.

David Andre and Astro Teller. Evolving Team Darwin United. *Lecture Notes in Computer Science*, 1604:346, 1999.

Brenna Argall, Sonia Chernova, Manuela M. Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.

Christopher G. Atkeson and Stefan Schaal. Robot learning from demonstration. In *Proc. 14th International Conference on Machine Learning*, pages 12–20. Morgan Kaufmann, 1997.

Aijun Bai, Feng Wu, and Xiaoping Chen. Online planning for large MDPs with MAXQ decomposition. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 3*, pages 1215–1216. International Foundation for Autonomous Agents and Multiagent Systems, 2012.

Aijun Bai, Feng Wu, and Xiaoping Chen. Towards a principled solution to simulated robot soccer. In *RoboCup 2012: Robot Soccer World Cup XVI*, pages 141–153. Springer, 2013.

Bram Bakker. Reinforcement learning with Long Short-Term Memory. In *Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, December 3-8, 2001, Vancouver, British Columbia, Canada]*, pages 1475–1482. MIT Press, 2001.

Samuel Barrett and Peter Stone. Cooperating with unknown teammates in complex domains: A robot soccer case study of ad hoc teamwork. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 2010–2016, January 2015.

Samuel Barrett, Peter Stone, Sarit Kraus, and Avi Rosenfeld. Learning teammate models for ad hoc teamwork. In *AAMAS Adaptive Learning Agents (ALA) Workshop*, June 2012.

Samuel Barrett. *Making Friends on the Fly: Advances in Ad Hoc Teamwork*. PhD thesis, The University of Texas at Austin, Austin, Texas, USA, December 2014.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

Marc G. Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Rémi Munos. Unifying count-based exploration and intrinsic motivation. *CoRR*, abs/1606.01868, 2016.

Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

Nuttapong Chentanez, Andrew G. Barto, and Satinder P. Singh. Intrinsically motivated reinforcement learning. In L. K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 1281–1288. MIT Press, 2005.

Bruno Castro da Silva, Gianluca Baldassarre, George Konidaris, and Andrew G. Barto. Learning parameterized motor skills on a humanoid robot. In *ICRA*, pages 5239–5244. IEEE, 2014.

Chelsea Finn and Sergey Levine. Deep visual foresight for planning robot motion. *CoRR*, abs/1610.00696, 2016.

Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. Learning visual feature spaces for robotic manipulation with deep spatial autoencoders. *CoRR*, abs/1509.06113, 2015.

Chelsea Finn, Sergey Levine, and Pieter Abbeel. Guided cost learning: Deep inverse optimal control via policy optimization. *CoRR*, abs/1603.00448, 2016.

Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate to solve riddles with deep distributed recurrent q-networks. *CoRR*, abs/1602.02672, 2016.

Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. *CoRR*, abs/1605.06676, 2016.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

Amy Greenwald and Keith Hall. Correlated-Q learning. In *In AAAI Spring Symposium*, pages 242–249. AAAI Press, 2003.

Shixiang Gu, Timothy P. Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. *CoRR*, abs/1603.00748, 2016.

Xiaoxiao Guo, Satinder Singh, and Richard L Lewis. Reward mapping for transfer in long-lived agents. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2130–2138. Curran Associates, Inc., 2013.

Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning. In Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3338–3346. Curran Associates, Inc., 2014.

Roland Hafner and Martin Riedmiller. Reinforcement learning in feedback control. *Machine Learning*, 84(1-2):137–169, 2011.

Matthew Hausknecht and Peter Stone. Learning powerful kicks on the Aibo ERS-7: The quest for a striker. In *Proceedings of the RoboCup International Symposium 2010*. Springer Verlag, 2010.

Matthew J. Hausknecht and Peter Stone. Deep recurrent Q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015.

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

Matthew Hausknecht and Peter Stone. On-policy vs. off-policy updates for deep reinforcement learning. In *Deep Reinforcement Learning: Frontiers and Challenges, IJCAI Workshop*, July 2016.

Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general Atari game playing. In *IEEE Transactions on Computational Intelligence and AI in Games*, 2013.

Nicolas Heess, Gregory Wayne, David Silver, Tim Lillicrap, Tom Erez, and Yuval Tassa. Learning continuous control policies by stochastic value gradients. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2944–2952. Curran Associates, Inc., 2015.

Todd Hester and Peter Stone. Intrinsically motivated model learning for developing curious robots. *Artificial Intelligence*, May 2015.

Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

Geoffrey E. Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Process. Mag.*, 29(6):82–97, 2012.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

Junling Hu and Michael P. Wellman. Nash Q-learning for general-sum stochastic games. *J. Mach. Learn. Res.*, 4:1039–1069, December 2003.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

Shivaram Kalyanakrishnan, Yaxin Liu, and Peter Stone. Half field offense in RoboCup soccer: A multiagent reinforcement learning case study. In Gerhard Lakemeyer, Elizabeth Sklar, Domenico Sorenti, and Tomoichi Takahashi, editors, *RoboCup-2006: Robot Soccer World Cup X*, volume 4434 of *Lecture Notes in Artificial Intelligence*, pages 72–85. Springer Verlag, Berlin, 2007.

Andrej Karpathy, Justin Johnson, and Fei-Fei Li. Visualizing and understanding recurrent networks. *arXiv preprint*, 2015.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

Nate Kohl and Peter Stone. Machine learning for fast quadrupedal locomotion. In *The Nineteenth National Conference on Artificial Intelligence*, pages 611–616, July 2004.

Jelle R. Kok and Nikos Vlassis. Mutual modeling of teammate behavior. Technical Report IAS-UVA-02-04, Informatics Institute, University of Amsterdam, The Netherlands, August 2002.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Proc. NIPS*, pages 1097–1105, Lake Tahoe, Nevada, USA, 2012.

Sascha Lange, Martin A. Riedmiller, and Arne Voigtländer. Autonomous reinforcement learning on raw visual input data in a real world application. In *The 2012 International Joint Conference on Neural Networks (IJCNN), Brisbane, Australia, June 10-15, 2012*, pages 1–8, 2012.

Quoc Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg Corrado, Jeff Dean, and Andrew Ng. Building high-level features using large scale unsupervised learning. In *International Conference in Machine Learning*, 2012.

Y. L. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of IEEE*, 86(11):2278–2324, November 1998.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-End training of deep visuomotor policies. *CoRR*, abs/1504.00702, 2015.

Sergey Levine. *Motor Skill Learning with Local Trajectory Methods*. PhD thesis, Stanford University, Stanford, CA, USA, 2014.

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *ArXiv e-prints*, September 2015.

Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992. UMI Order No. GAX93-22750.

Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine Learning, Proceedings of the Eleventh International Conference, Rutgers University, New Brunswick, NJ, USA, July 10-13, 1994*, pages 157–163. Morgan Kaufmann, 1994.

Michael L. Littman. Friend-or-Foe Q-learning in General-Sum Games. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pages 322–328, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

Bingyao Liu, Satinder P. Singh, Richard L. Lewis, and Shiyin Qin. Optimal rewards in multiagent teams. In *ICDL-EPIROB*, pages 1–8. IEEE, 2012.

Bingyao Liu, S. Singh, R.L. Lewis, and Shiyin Qin. Optimal rewards for cooperative agents. *Autonomous Mental Development, IEEE Transactions on*, 6(4):286–297, December 2014.

Patrick MacAlpine, Mike Depinet, and Peter Stone. UT Austin Villa 2014: RoboCup 3D simulation league champion via overlapping layered learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI)*, January 2015.

Patrick MacAlpine, Mike Depinet, and Peter Stone. UT Austin Villa 2014: RoboCup 3D simulation league champion via overlapping layered learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI)*, volume 4, pages 2842–48, January 2015.

Warwick Masson and George Konidaris. Reinforcement learning with parameterized actions. *CoRR*, abs/1509.01644, 2015.

Francisco S. Melo and M. Isabel Ribeiro. *Q-Learning with Linear Function Approximation*, pages 308–322. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

Francisco S. Melo, Sean P. Meyn, and M. Isabel Ribeiro. An analysis of reinforcement learning with function approximation. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 664–671, New York, NY, USA, 2008. ACM.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

Karthik Narasimhan, Tejas Kulkarni, and Regina Barzilay. Language understanding for text-based games using deep reinforcement learning. *CoRR*, abs/1506.08941, 2015.

Sanmit Narvekar, Jivko Sinapov, Matteo Leonetti, and Peter Stone. Source task creation for curriculum learning. In *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2016)*, May 2016.

Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *In Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287. Morgan Kaufmann, 1999.

Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L. Lewis, and Satinder P. Singh. Action-conditional video prediction using deep networks in atari games. *CoRR*, abs/1507.08750, 2015.

Junhyuk Oh, Valliappa Chockalingam, Satinder P. Singh, and Honglak Lee. Control of memory, active perception, and action in Minecraft. *CoRR*, abs/1605.09128, 2016.

Liviu Panait and Sean Luke. Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434, 2005.

Emilio Parisotto, Lei Jimmy Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *CoRR*, abs/1511.06342, 2015.

Martin A. Riedmiller and Thomas Gabel. On experiences in a complex and competitive gaming domain: Reinforcement learning meets robocup. In *CIG*, pages 17–23. IEEE, 2007.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

Andrei A. Rusu, Sergio Gomez Colmenarejo, Çaglar Gülçehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation. *CoRR*, abs/1511.06295, 2015.

Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *CoRR*, abs/1606.04671, 2016.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438, 2015.

David Silver, Aja Huang, Christopher 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*, 529:484–503, 2016.

S. Singh, R. L. Lewis, A. G. Barto, and J. Sorg. Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Trans. on Auton. Ment. Dev.*, 2(2):70–82, June 2010.

Jonathan Sorg, Richard L Lewis, and Satinder P. Singh. Reward design via online gradient ascent. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2190–2198. Curran Associates, Inc., 2010.

Bradly C. Stadie, Sergey Levine, and Pieter Abbeel. Incentivizing exploration in reinforcement learning with deep predictive models. *CoRR*, abs/1507.00814, 2015.

Kenneth O. Stanley and Risto Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*, page 9, San Francisco, 2002. Morgan Kaufmann.

Peter Stone and Manuela Veloso. Layered learning. In Ramon López de Mántaras and Enric Plaza, editors, *Machine Learning: ECML 2000 (Proceedings of the Eleventh European Conference on Machine Learning)*, pages 369–381. Springer Verlag, Barcelona,Catalonia,Spain, May/June 2000.

Peter Stone, Patrick Riley, and Manuela Veloso. Defining and using ideal teammate and opponent models. In *iaai2000*, pages 1040–1045, 2000.

Peter Stone, Gal A. Kaminka, Sarit Kraus, and Jeffrey S. Rosenschein. Ad hoc autonomous agent teams: Collaboration without pre-coordination. In *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence*, July 2010.

Ashley Stroupe, Martin C. Martin, and Tucker Balch. Distributed sensor fusion for object position estimation by multi-robot systems. In *IEEE International Conference on Robotics and Automation, May, 2001*. IEEE, May 2001.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

Milind Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.

Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In Michael N. Huhns and Munindar P. Singh, editors, *Readings in Agents*, pages 487–494. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.

Laurens van der Maaten and Geoffrey E. Hinton. Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with Double Q-learning. *CoRR*, abs/1509.06461, 2015.

Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

Shimon Whiteson and Peter Stone. Concurrent layered learning. In Jeffrey S. Rosenschein, Tuomas Sandholm, Michael Wooldridge, and Makoto Yokoo, editors, *Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 193–200, New York, NY, July 2003. ACM Press.

Shimon Whiteson. *Adaptive Representations for Reinforcement Learning*. PhD thesis, Department of Computer Science, University of Texas at Austin, May 2007.

Daan Wierstra, Alex Foerster, Jan Peters, and Juergen Schmidthuber. Solving deep memory POMDPs with recurrent policy gradients, 2007.

Markus Wulfmeier, Peter Ondruska, and Ingmar Posner. Deep inverse reinforcement learning. *CoRR*, abs/1507.04888, 2015.

Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Proc. ECCV*, pages 818–833, Zurich, Switzerland, 2014.

Matthew D. Zeiler. ADADELTA: An adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

Chongjie Zhang and Victor Lesser. Coordinating multi-agent reinforcement learning with limited communication. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '13, pages 1101–1108, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems.

Pucheng Zhou and Huiyan Shen. Multi-agent cooperation by reinforcement learning with teammate modeling and reward allotment, 2011.