Deep Reinforcement Learning Variants of Multi-Agent Learning Algorithms

Alvaro Ovalle Castañeda

Master of Science

School of Informatics

University of Edinburgh

2016

Abstract

We introduce *Deep Repeated Update Q-Network* (DRUQN) and *Deep Loosely Coupled Q-Network* (DLCQN). Two novel variants of Deep Q-Network (DQN). These algorithms are designed with the intention of providing architectures that are more appropriate for handling interactions between multiple agents and robust enough to deal with the non-stationarity produced by concurrent learning. We approach this from two different fronts. DRUQN tries to address Q-Learning's tendency to favor the update of certain action-values which may lead to decreased performance in rapid changing environments. Meanwhile, DLCQN learns to decompose the state space into two: (1) states where it is sensible or necessary to act independently and (2) those where acting in coordination with another agent may lead to a better outcome. We use *Pong* as testing environment and compare the performance of DRUQN, DLCQN and DQN on different competitive and cooperative experiments. The results demonstrate that for some tasks DLCQN and DRUQN outperform DQN which hints at the necessity to develop and using architectures capable of coping with richer and more complex dynamics.

Acknowledgements

This is the culmination of a very intensive but fruitful year. First of all, I am especially grateful to Dr. Henry Thompson and Dr. Sherief Abdallah for giving me the opportunity to participate in this extremely rewarding project. I would also like to thank them for their very valuable supervision and all the advise they provided me.

I would like to thank Santander Group for granting me an scholarship. I am very thankful for their contribution which helped to alleviate some of the financial burdens that come with studying abroad.

Finally, thanks to my family for their unconditional support, love and motivation. None of this would have been possible without them.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Alvaro Ovalle Castañeda*)

Contents

List of Figures vii

List of Tables ix

List of Algorithms x

1 Introduction 2

1.1 Overview . . . 2

1.2 Outline . . . 3

2 Preliminaries 5

2.1 Markov Decision Processes . . . 5

2.1.1 Solving Markov Decision Processes . . . 7

2.1.2 Partially Observable Markov Decision Processes . . . 8

2.1.2.1 Decentralized Markov Decision Processes . . . 10

2.2 Reinforcement Learning . . . 10

2.2.1 Q-Learning . . . 12

2.3 Approximating Action Value Functions . . . 13

2.3.1 Deep Learning . . . 14

2.3.1.1 Deep Q-Learning . . . 15

2.3.1.2 Other Extensions . . . 18

3 Multi-Agent Reinforcement Learning 19

3.1 Repeated Update Q-Learning . . . 21

3.1.1 Q-Learning Overestimation . . . 21

3.1.2 Addressing Overestimation with RUQL . . . 22

3.1.3 Related Work . . . 24

3.2 Loosely Coupled Learning . . . 25

3.2.1 Overview . . . 25
3.2.1.1 Agent independence . . . 25
3.2.1.2 Determining Independence Degree ξ_i^k . . . 25
3.2.1.3 Coordinated Learning . . . 27
3.2.2 Related Work . . . 29

4 Extending to Multi-Agent Deep Reinforcement Learning 31
4.1 Deep Repeated Update Q-Learning . . . 31
4.2 Deep Loosely Coupled Q-Learning . . . 33
4.2.1 Determining a Single Independence Degree ξ_k . . . 33
4.2.2 Approximating the Local and Global Q-functions . . . 35
4.3 Prior Research . . . 37

5 Methodology 39
5.1 Experimental Setup . . . 39
5.1.1 1-Player Control . . . 40
5.1.2 2-Player Control . . . 40
5.1.3 2-Player Control Mixed . . . 41
5.2 Architecture . . . 42
5.2.1 DLCQN Joint Network . . . 43
5.2.2 Training . . . 44
5.2.2.1 Parameters Specific to DLCQN . . . 44
5.3 Evaluation . . . 44

6 Emprical Evaluation 46
6.1 1-Player Control . . . 46
6.2 2-Player Control . . . 47
6.3 2-Player Control Mixed . . . 49
6.3.1 DQN vs DLCQN . . . 49
6.3.2 DQN vs DRUQN . . . 50
6.3.3 DRUQN vs DLCQN . . . 51

7 Discussion 53
7.1 Summary . . . 53
7.2 Observations . . . 55
7.2.1 Generality . . . 55

7.2.2 Non-Stationarity 56
7.2.3 DRUQN or DLCQN 57
7.3 Future Work 58

Bibliography **60**

A DRUQN Update Rule **66**

B DRUQN Approximation with ADAM **67**

C Full List of Network Parameters **69**

List of Figures

2.1 The reinforcement learning sensory-action loop (Sutton and Barto, 1998). 11
2.2 The architecture of the Deep Q-Network. The input of the network consists of the four most recent frames from the game. The space is subsequently transformed through three convolutional and two fully connected layers. The network outputs one of 18 possible actions (Mnih et al., 2015). 16

5.1 Network architecture for DRUQN and DLCQN (local). The input consists on the four most recent frames. The input is then transformed by two hidden convolutional+max-pool layers. Then followed by another hidden fully connected hidden layer. The output layer contains the Q-value estimated for each of the three possible actions. 42
5.2 Left: The paddle of the agents are not aligned in parallel and it is considered that they are not in each other's field of vision. Only the four most recent frames are sent as input to the agent local network. Right: Here, the paddles are aligned and information about the previous four frames can be shared between the agents. If the agent decides to act based on the shared information, eight frames are passed to the global shared network. 44

6.1 Performance comparison between DQN, DRUQN and DLCQN in the 1-Player control task. Top left: Average rewards of DQN and DLCQN. Top right: Average rewards of DQN and DRUQN. Bottom: Average maximal Q-values. 46
6.2 Performance comparison between DQN, DRUQN and DLCQN in the 2-Player control cooperative task. Left: Average paddle bouncing for DQN and DRUQN. Right: Average paddle bouncing between DQN and DLCQN. 47

6.3 Convergence in the 2-Player control cooperative task. Top left: DQN networks. Top right: DRUQN networks. Bottom: DLCQN shared and local networks. 48
6.4 DQN vs DLCQN competitive mode. Left: Q-value convergence. Right: Reward when competing against each other. 49
6.5 DQN vs DRUQN competitive mode. Left: Q-value convergence. Right: Reward when competing against each other. 50
6.6 DRUQN vs DLCQN competitive mode. Left: Q-value convergence. Right: Reward when competing against each other. 51

List of Tables

5.1 Reward structure in Pong. Competitive game mode with a single network controlled player. 40
5.2 Reward structure in Pong in a cooperative coordination game mode. Both players are controlled by their own network. 41
5.3 Reward structure in Pong in a competitive game mode. Both players are controlled by their own network. 42

6.1 DQN, DRUQN and DLCQN performance comparison in the 1-Player control task. 47
6.2 DQN, DRUQN and DLCQN performance comparison in the 2-Player control cooperative task. 48
6.3 DQN vs DLCQN competitive mode. First row shows the results when DQN performed the best against DLCQN. The second row is the best performing epoch of DLCQN against DQN. 49
6.4 DQN vs DRUQN competitive mode. First row shows the results when DQN performed the best against DRUQN. The second row is the best performing epoch of DRUQN against DQN. 50
6.5 DRUQN vs DLCQN competitive mode. First row shows the results when DRUQN performed the best against DLCQN. The second row is the best performing epoch of DLCQN against DRUQN. 51

List of Algorithms

1 Policy Iteration 8
2 Value Iteration 8
3 Q-Learning 13
4 Deep Q-Learning 17
5 Repeated Update Q-Learning: Intuition 23
6 Repeated Update Q-Learning: Practical Implementation 23
7 Independence Degree Adjustment 27
8 Coordinated Learning for agent i 28
9 Deep Repeated Update Q-Learning 33
10 Independence Degree Adjustment for agent k (DLCQL) 34
11 Deep Loosely Coupled Q-Learning for agent k 36

Chapter 1

Introduction

1.1 Overview

Humans and other animals are actively engaged with their surroundings. They take actions that carry consequences, affecting their environment and future observations. Frequently, these actions are purposeful and they can be understood as a series of sequential decision making steps leading to an outcome. Reinforcement learning (RL) is concerned with the study of goal directed behavior. Basically, in RL an *agent* is embedded in an environment. From there we then consider what actions need to be selected in order to achieve certain outcome. An agent facing a new task learns about the effect of its actions in the environment, by observing how it changes and contrasting how much these observations differ with respect to its expectations.

A significant portion of the RL literature has focused on single agents, however most realistic situations involve the presence of multiple agents. Some tasks can only be solved or conceived by the interaction between different actors. Multi-agent reinforcement learning (MARL) incorporates advancements from single agent RL but poses additional challenges. For example, the definition of suitable collective and/or individual goals (Agogino and Tumer, 2005; Buşoniu et al., 2008), how to deal with heterogeneous learners (Panait and Luke, 2005) or the design of compact representations that can scale to a large number of agents (Buşoniu et al., 2008). This dissertation is concerned with *non-stationarity*, which is one of the main difficulties in the study of MARL. Non-stationarity occurs because the interaction of multiple agents constantly reshapes the environment. Unlike in single agent RL, where the agent is observing only the effect of its own actions. In MARL, agents are interacting and learning simultaneously in the same environment. An agent begins to associate not only its

action to certain outcomes but also to the behavior observed in other agents. At the same time the other agents also start adjusting their own behavior. Consequently associations that were learned by the agent in the past may no longer hold in the present. Thus in a non-stationary environment the estimation of potential benefits of an action can become obsolete.

Several RL algorithms have been developed to estimate the value of an action in the context of a specific situation. Among them Q-Learning (Watkins, 1989; Watkins and Dayan, 1992) stands out for its popularity, for being intuitive and easy to implement. Q-Learning, as it is the case with other RL algorithms, is restricted to operate in small state spaces. This limitation has been overcome with the use of function approximators (Kaelbling et al., 1996). Recently, Q-Learning was integrated with a deep convolutional neural network to conceive the Deep Q-Network (DQN) architecture (Mnih et al., 2013, 2015). The DQN was trained to learn Atari 2600 games by receiving only the pixels from the screen. Some of the games DQN was tested on, featured simultaneous multi-player mode. Q-Learning has been used in multi-agent scenarios in the past. However it was not designed for the non-stationary environments that result from the interaction between multiple agents. Furthermore, the theoretical convergence guarantees that Q-Learning provides, do not extend to non-stationary environments.

In this dissertation, we introduce two novel variants of DQN specifically intended to handle the non-stationarity inherent to multi-agent environments. Our research is then concerned with analyzing and comparing the performance of these variants to the original DQN in a multi-agent context. The first algorithm, *Deep Repeated Update Q-Network* (DRUQN) is based on the work by Abdallah and Kaisers (2013, 2016). It tries to address an issue in the way Q-Learning estimates the value of an action. The second algorithm, *Deep Loosely Coupled Q-Network* (DLCQN), inspired by Yu et al. (2015), assumes that an agent is not capable of observing the full information content of the environment. Therefore an agent has to learn under which circumstances it has to act independently and when in coordination with other agents or the information they provide.

1.2 Outline

The rest of the dissertation is structured as follows. In the next chapter we provide a brief introduction of the theoretical foundations of Markov Decision Processes, a mathematical framework necessary to formalize our notions of sequential decision

making. Partially Observable Markov Decision Processes and Decentralized Markov Decision Processes are also described as extensions to subclasses of problems where the assumptions made by the Markov Decision Processes are not enough or do not hold. In addition, the underlying notions of Reinforcement Learning are provided as well as how it is adapted to deal with large state spaces. In addition the Deep Q-Network algorithm is described. Chapter 3 gives an overview of the issues afflicting the action value estimation in Q-Learning. Then we continue with Repeated Update Q-Learning, and coordinated learning algorithms which form the basis for the novel variants introduced in Chapter 4. In Chapter 5 we describe the methodology, architecture and experiments considered for the comparison of the different algorithms. Chapter 6 presents the results and experimental findings. Finally, Chapter 7 discusses the results, limitations and potential avenues of further research.

Chapter 2

Preliminaries

2.1 Markov Decision Processes

Markov Decision Processes (MDP) are a mathematical framework used to represent sequential decision making in situations where outcomes are uncertain. They have been widely used in optimization, economics, control and robotics. For the work here presented, they offer the possibility to model the dynamics of an agent interacting with its environment. More formally, an MDP is defined by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$ where,

- $\mathcal{S}$ is a set of states $s_i \in \mathcal{S} | i = 1...n$
- $\mathcal{A}$ is a set of actions $a_i \in \mathcal{A} | i = 1...m$
- $\mathcal{P}$ is the transition matrix and contains the probabilities of going from state s to state s' if selecting action a.
  $\mathcal{P}^a_{ss'} = \mathbb{P}[\mathcal{S}_{t+1} = s' | \mathcal{S}_t = s, \mathcal{A}_t = a]$
- $\mathcal{R}$ is the reward function received after selecting action a in state s.
  $\mathcal{R}^a_s = \mathbb{E}[\mathcal{R} | \mathcal{S}_t = s, \mathcal{A}_t = a]$

From above, it is now established that an MDP can be used to model the process by which an agent will receive a reward depending on the action, and current state of the environment. The state can be understood as the information that is available for the agent. MDPs simplify the computation of the transition distribution from the current state to the next one by satisfying the Markov property. This means that an MDP is memoryless and history independent. A future state will only be determined by the current state ignoring or treating as irrelevant all previous states that happened before. Thus,

$$\mathbb{P}[\mathcal{S}_{t+1} = s' | \mathcal{S}_t = s] = \mathbb{P}[\mathcal{S}_{t+1} = s' | \mathcal{S}_1 = s_1 ... \mathcal{S}_t = s_t]$$

The current state is assumed as sufficient statistic to predict the future as it captures all necessary information, this implies that an MDP assumes full observability of the environment at any given state. If taking into account an action at a given state the Markov property is reflected as,

$$\mathbb{P}[\mathcal{S}_{t+1} = s' | \mathcal{S}_t = s, \mathcal{A}_t = a] = \mathbb{P}[\mathcal{S}_{t+1} = s' | \mathcal{S}_1 = s_1, \mathcal{A}_1 = a_1 ... \mathcal{S}_t = s_t, \mathcal{A}_t = a_t]$$

MDPs evaluate and use policies to decide what action to take at each state. A policy π is a plan or a sequence of actions and define the behavior of the system. In its deterministic form a policy is given by $\pi : \mathcal{S} \to \mathcal{A}$ by contrast, a stochastic policy is a distribution over possible actions at a given state where $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$. In the stochastic case the transition function that describes the probability to move from state s to state s' is,

$$\mathcal{P}^{\pi}_{s,s'} = \sum_{a \in \mathcal{A}} \mathbb{P}[s' | s, a] \pi(s, a)$$

In order to solve an MDP an optimal policy π^* has to be found. From the perspective of the agent, this translates into finding a sequence of actions from the initial to the terminal state, that maximizes a long term measurement or reward. Accordingly, the reward function based on a stochastic policy is,

$$\mathcal{R}^{\pi}_{s} = \sum_{a \in \mathcal{A}} \mathcal{R}^{a}_{s} \pi(s, a)$$

The total reward that an MDP obtains from time t onwards is the expectation of accumulated rewards,

$$\mathcal{R}_t = \mathbb{E}[\sum_{t}^{\infty} r_t]$$

However if a problem has no defined termination, the sum becomes infinite as $t \to \infty$. In order to overcome this issue and make it tractable, a finite time horizon is defined. In some cases this may not be possible therefore a preferable alternative is to introduce a discount factor γ, where $\gamma \in [0, 1]$. The discount factor will determine the weight of future rewards in relation to how far they are into the future by decaying exponentially. The more γ is close to 1 the more it will take into consideration future rewards. On

the other hand, if γ is close to 0, the focus is on immediate rewards. This may lead a system to miscalculate the future. An MDP that is discounted is defined by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$. The total reward in its discounted form becomes,

$$\mathcal{R}_t = \mathbb{E}[r_t + \gamma r_{t+1} + ...] = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t] \tag{2.1}$$

Equation 2.1 can also be decomposed as,

$$\mathcal{R}_t = \mathbb{E}[r_t + \gamma \mathcal{R}_{t+1}] \tag{2.2}$$

2.1.1 Solving Markov Decision Processes

Although MDPs are able to follow different policies, it was mentioned earlier that we ideally want to find policies that maximize the cumulative rewards as much as possible. For every MDP there is an optimal policy π^* and consequently no other policy will do better. To differentiate among policies a state value function is needed as a way to evaluate and measure how well a policy performs,

$$\mathcal{V}^{\pi}(s) = \mathbb{E}_{\pi}[\mathcal{R}_t | \mathcal{S} = s_t] \tag{2.3}$$

It will denote the value $\mathcal{V}$ as the expected total discounted reward from the current state s when following policy π. From equation 2.1 it is known that the total discounted reward is the expected immediate reward and all future discounted rewards when following the same policy π. Accordingly if following an optimal policy π^* the state value $\mathcal{V}^*$ obtained at each state is guaranteed to be the best possible. From equation 2.2 and equation 2.3, we obtain the Bellman equation as follows (Bellman, 1954, 1956),

$$\begin{aligned} \mathcal{V}^{\pi}(s) &= \mathbb{E}_{\pi}[r_t + \gamma \mathcal{V}^{\pi}(s')] \\ &= r_t + \sum_{s' \in \mathcal{S}} \gamma \mathcal{P}^{\pi}(s'|s) \mathcal{V}^{\pi}(s') \end{aligned}$$

Thus a policy is optimal when the state value function satisfies the Bellman optimality equation,

$$\mathcal{V}^{\pi}(s) = \max_{a \in \mathcal{A}} r_t(s,a) + \sum_{s' \in \mathcal{S}} \gamma \mathcal{P}^{\pi}(s'|s,a) \mathcal{V}^{\pi}(s') \tag{2.4}$$

Knowing the optimal value then, the optimal policy corresponds to,

$$\pi^*(s) = arg\max_{a\in\mathcal{A}} r_t(s,a) + \sum_{s'\in\mathcal{S}} \gamma\mathcal{P}^{\pi}(s'|s,a)\mathcal{V}^*(s')$$

To solve the system of optimal equations, iterative techniques such as dynamic programming must be used. Policy iteration (Howard, 1960) and value iteration (Bellman, 1956) are included among the most well known methods.

Algorithm 1 Policy Iteration

Initialize π to arbitrary π^0
for $i = 1...$ **do**
  Solve: $\mathcal{V}^{i-1} = r_t + \sum_{s'\in\mathcal{S}} \gamma\mathcal{P}^{\pi}(s'|s)\mathcal{V}^{\pi}(s')$
  for $s \in \mathcal{S}$ **do**
    $\pi^i(s) = arg\max_{a\in\mathcal{A}} r_t(s,a) + \sum_{s'\in\mathcal{S}} \gamma\mathcal{P}^{\pi}(s'|s,a)\mathcal{V}^{\pi}(s')$
  end for
  $\pi = \pi^i$
end for

In the case of policy iteration, an optimal policy is found in at most $|\mathcal{A}|^{|\mathcal{S}|}$ steps. The algorithm finishes once the policy stops improving (Algorithm 1). In the second algorithm, for value iteration, the state value functions are used to indirectly obtain a good policy simply by updating Equation 2.4. Since it is uncertain when the state value functions can converge to an optimal $\mathcal{V}^*$, different forms of stopping criteria have been proposed (Puterman, 1994; Sutton and Barto, 1998).

Algorithm 2 Value Iteration

Initialize to arbitrary $\mathcal{V}^0$
for $i = 1...$ **do**
  for $s \in \mathcal{S}$ **do**
    $\mathcal{V}^i(s) = \max_{a\in\mathcal{A}} r_t(s,a) + \sum_{s'\in\mathcal{S}} \gamma\mathcal{P}(s'|s,a)\mathcal{V}^{i-1}(s')$
  end for
end for

2.1.2 Partially Observable Markov Decision Processes

A crucial aspect in solving MDPs is to assume full observability. This supposes that accurate and fully visible information is available at every time step and that this information is contained in the current state. However in many complex or real world

scenarios this assumption is unfeasible. For multi-agent environments for instance, it is easy to imagine situations where an agent may rely on the knowledge from other agents in order to make a decision. Even with single agents, in very large or constrained environments, the access to information may be limited to a local region. Partially Observable Markov Decision Processes (POMDP) (Åström, 1965) is an extension to classical MDPs designed to account for situations where the information that can be accessed at a given state is incomplete. A POMDP is defined by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{O}, \mathcal{Z} \rangle$ where $\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}$ follow from the original elements of an MDP. For the new elements $\mathcal{O}$ and $\mathcal{Z}$,

- $\mathcal{O}$ is a set of observations.
- $\mathcal{Z}$ is an observation function describing the probability of observing o' if action a is executed and the environment transitions to unobservable state s'.
  $\mathcal{Z}^{a}_{s'o} = \mathbb{P}[\mathcal{O}_{t+1} = o' | \mathcal{S}_{t+1} = s', \mathcal{A}_t = a]$

This extension to MDPs however increase the complexity required to solve them. POMDPs stop being Markovian, as the current state cannot be inferred solely by the set of observations from a process. Frequently POMDPs will rely on the whole sequential history of actions, observations and rewards $\mathcal{H}_t = a_0, o_2, r_1 ... a_{t-1}, o_t, r_t$ that have been observed in order to construct a belief state $b(s)$. Thus these belief states summarize the particular experience by representing probability distributions over the states of the system. Belief states will act as sufficient statistics allowing POMDPs to satisfy the Markov property. The Bellman equation with belief states can be obtained as,

$$\mathcal{V}^{\pi}(b) = \sum_{s \in \mathcal{S}} b(s) r_t + \sum_{o \in \mathcal{O}} \gamma \mathcal{P}^{\pi}(o|b) \mathcal{V}^{\pi}(b_o)$$

Algorithms similar to those used by MDPs can be applied to maximize the outcome, with the main difference that POMDPs will take into consideration the current belief as opposed to a state. A potential problem occurs when the number of states of the problem is large. Finding a good policy becomes intractable as evaluating state value function over the space of beliefs is unmanageable due to their high dimensionality and continuity (N. Roy, 2003). In order to solve them, some properties of the problem can be exploited to simplify it. However in many instances to solve the POMDPs we will have to approximate the belief states. Several techniques exist such as QMDP (Littman et al., 1995), Monte Carlo (Thrun, 2000), Augmented-MPDs (N. Roy and Thrun, 1999) and belief state compression (N. Roy, 2003).

2.1.2.1 Decentralized Markov Decision Processes

In multi-agent domains, an agent may not only depend on the information it has gathered about its environment. It will also be influenced by the choices of other agents. Naturally, these problems are partially observable. Decentralized Partially Observable Markov Decision Processes (Dec-POMDP) (Bernstein et al., 2000) have been developed as an extension of POMDPs to address situations where agents can exploit levels of coordination among them. Using this framework a process may act as dependent or as independent depending on the degree of coordination required by the agents. In a similar way to POMDPs, a Dec-POMDP is defined by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{O}, \mathcal{Z} \rangle$ where,

- $\mathcal{A} = {}_{i \in I}\mathcal{A}_i$, is the set of joint actions, where $\mathcal{A}_i$ is the set of actions from agent i.
- Accordingly the transition function $\mathcal{P}^{\vec{a}}_{ss'}$ and the reward function $\mathcal{R}^{\vec{a}}_{s}$ consider the joint action $\vec{a} = \langle a_1, ..a_I \rangle$.
- $\mathcal{O} = {}_{i \in I}\mathcal{O}_i$, is the set of joint observations.
- $\mathcal{Z}$ is an observation function describing the probability of observing joint $\vec{o}$ given joint action $\vec{a}$ leading to state s'.
  $\mathcal{Z}^{\vec{a}}_{s'\vec{o}} = \mathbb{P}[\vec{o}|s', \vec{a}]$

In some cases the full observability of a state can be assumed by combining the information from each member. In particular for some of the tasks presented in this dissertation, a 2-agent factored Dec-MDP (Allen and Zilberstein, 2009; Roth et al., 2007) is used to model multi-agent decision making. Thus it is said that a Dec-MDP is a Dec-POMDP that is jointly fully observable. Thus state $\mathcal{S}$ can be retrieved by the individual observations $\mathcal{O}_i$ of each agent. The tuple can be simplified to $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$ by considering $\mathcal{O}_i$ as the state space of agent i, $\mathcal{S}_i$. The system state can the be collectively determined (Pynadath and Tambe, 2002) and defined as $\mathcal{S} = \mathcal{S}_0 \times \mathcal{S}_i \times ... \times \mathcal{S}_I$, where $\mathcal{S}_0$ could be a portion of the state that is shared among all agents and $\mathcal{S}_i$ the local state of agent i. The local state of an agent at a given time then is $\vec{s}_i = \langle s_0, s_i \rangle$.

2.2 Reinforcement Learning

Previously we discussed how MDPs can be solved through the use of dynamic programming techniques. By iterating, evaluating and calculating state value functions it is possible to find optimal policies. Dynamic programming techniques assume knowledge

of the model and the environment since they require the reward function and the transition probabilities. This is possible only in ideal conditions. For the large majority of the cases these quantities must be discovered.

Reinforcement learning (RL) offers a framework to represent interactive based learning. It is inspired by insights from classical and operant conditioning experiments where the subjects learn certain behavior based on its outcome. Recent scientific evidence in the identification of dopaminergic neurons as well as their role in coding error signals and event prediction, have also helped to consolidate RL as modeling tool in neuroscience (Dayan and Niv, 2008; Schultz et al., 1997). Essentially in RL an agent discovers the structure of an unknown environment by interacting with it by trial and error. At each time step, the agent performs an action. It then receives feedback about it which comes in the form of updated perceptual information and rewards.

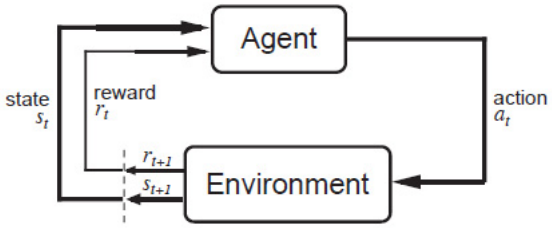

Figure 2.1 The reinforcement learning sensory-action loop (Sutton and Barto, 1998).

For solving MDPs, RL algorithms can be used for obtaining optimal policies without previous knowledge of the model (i.e. transition and reward functions). RL explores the state space and learns a policy from observing the outcomes. Depending on the algorithm choice there are two ways to proceed: (1) learn a model of the environment or (2) use a policy implicitly by estimating only value functions. Due to its appealing generative properties model-based RL may be preferable for planning and for data efficiency, however acquiring the model is computationally expensive. For model-free algorithms it is enough to know the state and the actions the agent has at its disposal. This is one of the reasons that has contributed to the popularity of model-free RL in the literature.

A fundamental difference with respect to value and policy iteration is that in those techniques an MDP visits each state to update their respective value. By contrast RL algorithms sample the environment updating the information of only those states that are experienced. This leads to the *exploration - exploitation* dilemma. An agent should use its knowledge to select the most profitable actions (exploitation), and at the same time

attempt to discover better courses of action and areas of the state space. Exploration is specially necessary when facing a new environment since a correct value estimation depends on the knowledge of it. A balance between exploration and exploitation is required. Most common strategies rely on controlling a parameter that regulates the balance between exploration and exploitation. For example, a temperature parameter τ using a Boltzmann distribution or ε in ε-greedy. In the case of ε-greedy, an exploratory action is selected with probability ε and a greedy action with 1-ε. There is not a final solution for this dilemma and more sophisticated exploration approaches have been proposed (Abel et al., 2016; Asmuth et al., 2012; Jung and Stone, 2012; Osband et al., 2014).

2.2.1 Q-Learning

Several reinforcement learning methods such as SARSA (Rummery and Niranjan, 1994), actor-critic variants (Barto et al., 1983), REINFORCE (Williams, 1992) and other policy gradient algorithms (Silver et al., 2014) have been developed throughout the years. In this section, Q-Learning is introduced as a foundation of subsequent algorithms presented in this work. Q-learning (Watkins, 1989; Watkins and Dayan, 1992) is a popular model-free RL algorithm noted for its simplicity. A Q-function or action value function can be expressed as,

$$Q_\pi(s,a) = \mathbb{E}_\pi[\mathcal{R}_t|\mathcal{S} = s_t, \mathcal{A} = a_t]$$

Although similar to the value function from Equation 1.4, it is observed that there is an explicit consideration of the action being selected. In other words, a Q-value measures the quality or the value of an action at a given state. Correspondingly,

$$Q(s,a) = r + \gamma \max_a Q(s',a') \tag{2.5}$$

Since there is no assumption about prior knowledge of a model, the algorithm will sample the state and action space, as in lines 4-5 of Algorithm 3. Then the Q-values are updated according to what has been observed,

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_a Q(s',a') - Q(s,a)] \tag{2.6}$$

Where α is the learning rate that controls the amount that the values are updated. The learning rule is said to use temporal difference (Sutton, 1988). It involves refining

Algorithm 3 Q-Learning

Initialize to arbitrary $Q(s,a)$
Observe current state s
repeat
    Select an action a according to policy $\pi(s,a)$
    Execute action a
    Observe reward r and next state s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_a Q(s',a') - Q(s,a)]$
until Termination

the values from previous estimations. The process is repeated until the predictions start matching the observations. Given certain conditions such as a properly decaying learning rate (Kaelbling et al., 1996), specific environments and a finite MDP, Q-learning will converge and find the optimal values $\lim_{t\to\infty} Q(s,a) = Q^*(s,a)$. Q-learning is also considered an *off policy* algorithm because the Q-values are estimated assuming that for the next state a greedy policy will be followed regardless of the actual policy that is being followed.

2.3 Approximating Action Value Functions

In their most basic form, dynamic programming and reinforcement learning techniques compute values belonging to a specific state of an MDP. Up to this stage, it has been assumed that all these states are known or can be listed beforehand. For instance, value and policy iteration sweep through the states to update their estimation. In the case of Q-learning we observe in equation 2.5, that a value is computed for each state-action pair. For both cases, value functions are stored and represented through lookup tables. Each state-action pair has a corresponding cell in this table from where the value can be retrieved or updated. However this type of representation is only amenable for a reduced subset of problems with discrete and small state spaces. From a computational perspective holding in memory the lookup table representing the state-action of a large MDP results impractical. Function approximation can be used to substitute the lookup table for problems dealing with continuous domains or where the number of states is unknown or very large. Besides making for a more compact representation, the main aspect that makes function approximation useful, is that it allows to generalize to unseen states based on what has been experienced previously. Instead of learning the value

of each cell individually, it learns the parameters of a model which in turn allows to exploit the similarity between states. The approximation can be performed through various methods such as decision trees (Pyeatt and Howe, 1998), linear combinations with the application of different bases, for instance polynomial (Lagoudakis and Parr, 2003), RBF or Fourier (Konidaris and Osentoski, 2008) and non-linear such as neural networks (Tesauro, 1995; Tsitsiklis and B. V. Roy, 1997). Considering the case of a linear combination, an action-value would be given by,

$$\hat{Q}(s,a,\mathbf{w}) = \sum_{i}^{n} w_i \phi_i(s,a) \tag{2.7}$$

Where $\hat{Q}(s,a,\mathbf{w}) \approx Q(s,a)$ is the approximation of a state, $\mathbf{w}$ is the vector of weights $\mathbf{w} = [w_1, ...w_n]$ which corresponds to the learnable parameters necessary to perform the approximation. Instead of updating directly a Q-value, these are the parameters that will be updated. This type of approximation (which also applies to non-linear) is easily differentiable which facilitates the update of the parameters with gradient descent. From equation 2.6, it is known that the temporal difference,

$$\alpha[r + \gamma \max_a Q(s',a') - Q(s,a)] \tag{2.8}$$

Measures the prediction error between the new estimate and the previous one. This allows to define a loss function to be minimized,

$$L(\mathbf{w}) = [r + \gamma \max_a Q(s',a',\mathbf{w}) - \hat{Q}(s,a,\mathbf{w})]^2 \tag{2.9}$$

Where $r + \gamma \max_a Q(s',a',\mathbf{w})$, is considered as the target. Then the loss function is differentiated with respect to $\mathbf{w}$ in order to obtain the gradient,

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \left[\left(r + \gamma \max_a Q(s',a',\mathbf{w}) - \hat{Q}(s,a,\mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{Q}(s,a,\mathbf{w}) \right] \tag{2.10}$$

To finally obtain the parameter update rule,

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \nabla_{\mathbf{w}} L(\mathbf{w}) = \mathbf{w} + \alpha \left[\left(r + \gamma \max_a Q(s',a',\mathbf{w}) - \hat{Q}(s,a,\mathbf{w}) \right) \phi_i(s,a) \right] \tag{2.11}$$

2.3.1 Deep Learning

Function approximation alleviates some of the issues associated to large state spaces. However, in order to solve an MDP, state representation would still rely on the researcher

choices. Features are commonly hand crafted and adapted to a specific problem. A considerable part of the success in solving a problem then is determined by how well the features capture the peculiar intricacies of the situation. This is a concern not particular only to reinforcement learning but one that affects every area of machine learning. In recent years, the application of deep learning principles has provided new avenues of research into tackling these fundamental issues. Deep learning consists on the construction of multiple layers of artificial neurons with the objective of automatically learning features from data (Fukushima, 1980; LeCun et al., 2015; Schmidhuber, 2015). The increase of computational resources in combination with a better understanding of the properties of the networks or the design of more efficient non-linear functions (Nair and G. E. Hinton, 2010) have sparked a series of developments. Convolutional networks, for instance, attempt to mimic some of the mechanisms behind simple receptive fields and have become state of the art in image recognition (Krizhevsky et al., 2012). Autoencoders allow to obtain more condensed representations of the original data (Bengio et al., 2007), LSTM recurrent networks, model short term dependencies (Hochreiter and Schmidhuber, 1997), and more recently adversarial networks are being used as generative models (Goodfellow et al., 2014). A large majority of success stories from deep learning have come from areas such as vision (Krizhevsky et al., 2012), speech recognition (G. Hinton et al., 2012; Mohamed et al., 2012) or language processing (Bordes et al., 2014; Collobert et al., 2011; Jean et al., 2014; Sutskever et al., 2014), the interest has also expanded into other areas, including reinforcement learning. This has lead to an increasing interest into how to integrate these approaches.

2.3.1.1 Deep Q-Learning

Neural networks and reinforcement learning have been used in conjunction in the past (Riedmiller, 2000; Tesauro, 1995) as a mean to approximate action or state value functions. Positive results, however, had been limited to specific applications. The main problem observed was their instability. Networks would learn slowly or very inefficiently which in turn would lead to divergence in the values. Riedmiller (2005) introduced Neural Fitted Q-Iteration (NFQ) to try to address some of these issues. In principle, in previous approaches the divergence of values would be caused in part by updating the parameters on-line. NFQ instead, takes a concept called *experience replay* (Lin, 1992). It stores a set of previous transition experiences in tuples and then uses them to update the parameters off-line via batch gradient descent.

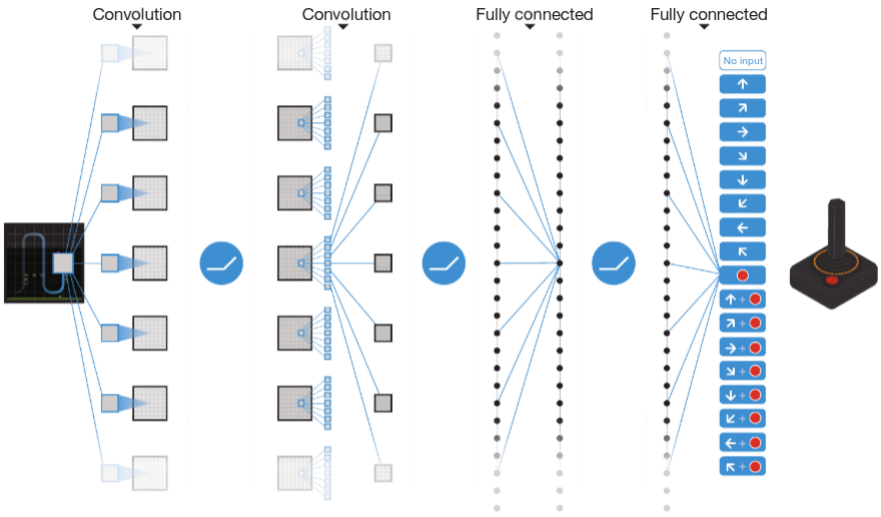

Figure 2.2 The architecture of the Deep Q-Network. The input of the network consists of the four most recent frames from the game. The space is subsequently transformed through three convolutional and two fully connected layers. The network outputs one of 18 possible actions (Mnih et al., 2015).

Mnih et al. (2013, 2015) extended further this approach. In their work they introduced an algorithm called Deep Q-Learning (DQN) and designed a convolutional deep neural network intended to play Atari 2600 games (Figure 2.2). The novel aspect of the work was that the same network architecture was used across the games, with only few changes between them (i.e. Space Invaders) or the inclusion of prior knowledge (e.g. reward clipping). For all games the input received by the network were the pixels from the screen. The network would then model the action-values particular to the screen scene, outputting an specific action to execute. Using only minimal pre-processing such as gray scaling and down sampling, the network was able to learn high level features that were relevant to learn how to play the games. The network achieved human or above human performance in 29 out of 49 games (Mnih et al., 2015).

As implied earlier, DQN uses experience replay to stabilize its behavior and deal with non-stationarity. The network randomly samples a minibatch from the stored memories. This is also intended to remove the correlations between the observations, assuming independence and smoothing their distribution, which is essential for the gradient based learning in which it relies on. DQN similarly to NFQ, keeps fixed the parameters from previous iterations in order to measure the error with respect to the updated parameters. In a second version of DQN, presented in Mnih et al. (2015), the model considers an additional network which serves as the target. The parameters in this network are held fixed and they are only updated after certain number of iterations

Algorithm 4 Deep Q-Learning

Initialize replay memory D to hold N transitions
Initialize action-value function Q with random weights
for episode=1...M **do**
  Initialize a sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for episode=1...T **do**
    Select a random action a_t with probability ε
    Otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t
    Observe reward r_t and next frame $x_t + 1$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform gradient descent on $(y_j - Q(\phi_j, a_j; \theta))^2$ wrt θ
    Every C steps set $\hat{Q} = Q$ ▷ If using an extra target network
  end for
end for

have passed. The purpose is to increase the robustness of the network.

The principles used by DQN follow closely those described in section 2.3. A loss function for each iteration i is defined as,

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)}[r + \gamma \max_a Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)]^2 \tag{2.12}$$

Where θ_i are the parameters of the network at iteration i, θ_i^- are the parameters of the target network and D is the replay memory that holds observations of the agent. The transitions (s, a, r, s') are randomly sampled from D. The expression is differentiated with respect to the parameters θ,

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)}[(r + \gamma \max_a Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i)] \tag{2.13}$$

And they are updated according to Equation 2.11,

$$\theta_{i+1} \leftarrow \theta_i + \alpha \nabla_{\theta_i} L(\theta_i) \tag{2.14}$$

2.3.1.2 Other Extensions

The publication of DQN has motivated a series of improvements to the original algorithm. Prioritized experience replay (Schaul et al., 2015) has been proposed as an alternative to uniform sampling from replay memory. The mechanism determines which transition to sample using stochastic prioritization and importance sampling. The transitions that have caused the most surprise in the past have a higher chance to be selected. The deep double Q-network presented by van Hasselt et al. (2015) tries to address overoptimistic estimation in Q-Learning. The study addresses it by decoupling the selection and the evaluation of actions. Wang et al. (2015) opt for altering the architecture of the network and divide it into two estimators, a state value function and an advantage function that determines the benefits from selecting specific action. In Sorokin et al. (2015), they extend DQN to LSTM networks to represent areas of attention. Other work extends beyond the application of deep neural networks to Q-learning as it is the case in Lillicrap et al. (2015), where they present an algorithm that generalizes to continuous spaces using deterministic policy gradients.

Chapter 3

Multi-Agent Reinforcement Learning

Most of real world problems are of a distributed nature or can be framed as such. They are based on the interaction between multiple parts or participants. Distributed environments can benefit from the communication and sharing information through imitation or teaching (Buşoniu et al., 2010). Furthermore decomposing large tasks into smaller ones or taking advantage of their decentralized properties allows for parallel and more expedited solutions. However, the complexity and richness in the dynamics of these kind of environments makes it problematic to design solutions that can encompass all potential scenarios. Ideally then, one should opt instead for designing agents that can be adaptable and robust enough to deal with a constantly changing environment. As we have seen, reinforcement learning provides an alternative to deal with such environments. RL agents learn from experience by observing their environment and the effect of their actions. Nonetheless the transition from single agent RL to multi-agent RL offers a series challenges.

The reward that the agent may receive will not only depend on its interaction with a passive environment. In multi-agent environments, it is intertwined with the actions made by the others. This first supposes that the type of tasks occurring has to be identified. A task can be *competitive* or *cooperative*. However, most environments will usually contain a combination of both. In the case of a competitive task, an agent tries to maximize its reward even if it affects other agents utilities. In the cooperative case, the success of the agent depends on the success of the other agents as well. Therefore there is a shared reward function that is maximized in conjunction with all agents. Defining a goal becomes complex because the rewards are correlated and cannot simply be maximized independently (Buşoniu et al., 2008, 2010). An agent trying to maximize its reward may not imply a collective maximization. In this sense, the optimal behavior

of an agent may not correspond to the most desirable joint policy. Simple scenarios have been studied from a game theoretic perspective (Panait and Luke, 2005). Under this optic single MDPs are extended to account for joint actions and are denominated as Markov or Stochastic Games (Littman, 1994). In most analyses it is assumed the agents are fixed at certain state which is then repeated multiple times. A stochastic game is then solved when a joint strategy finds a *Nash equilibrium* (Nash, 1950; Von Neumann and Morgenstern, 1944). Joint strategies satisfy this requirement when the action selected by the agent is the best possible action when considering the actions taken by the others. Thus changing to another action would offer no possible benefit.

Another challenge comes with the nature of the individual tasks assigned to each agent. Although the global goal may be cooperative, each agent could have its own individual decision making. It becomes fundamental to define what actions can be carried on with some degree of independence and which have to be taken in a coordinated manner. Similarly to single agent RL, where the structure of the environment is learned, in multi-agent RL part of learning said structure entails learning about the existence of other agents, their actions and their goals. Thus the level of awareness of the agent will have an impact in its performance (Tuyls and Weiss, 2012). Some tasks may not require any while for other tasks knowing information about the other agents is primordial.

One of the biggest open issues in multi-agent environments is how to deal with *non-stationarity*. A policy is optimal and *stationary* when it is the best possible policy and it remains fixed over time. Due to the dependence of the reward function on the actions taken by other agents, good policies at a given point could not be so in the future. They are only good policies in relation to what the other agents have learned at the time the policy is applied. The exploration-exploitation dilemma becomes even more relevant under these settings. Information gathering is not only important initially but has to be done with certain recurrence while at the same time being careful that it does not destabilize the agent or agents when an appropriate coordination is required.

When dealing with non-stationarity, the Markov property does not hold anymore. Consequently, theoretical convergence guarantees offered by single agent RL will not necessarily apply to most multi-agent RL problems. Further extensions to MDPs have been developed to account for multiple agents and convergence proofs have been provided (Bowling, 2000; Hu and Wellman, 1998; Littman, 1994, 2001a). However these proofs relax the assumptions to a large extent assuming that rewards and actions are observable or that agents learn equally. These assumptions turn out to be too strong and not very realistic (Tesauro, 2003).

In practice, convergence in most complex multi-agent problems tends to be empirically verified. In some cases single RL algorithms such as Q-Learning have been used with no modification (Claus and Boutilier, 1998; Crites and Barto, 1998; Tan, 1993). However several extensions to a multi-agent domain have been proposed for cooperative tasks (Kapetanakis and Kudenko, 2005; Lauer and Riedmiller, 2000; Littman, 2001b), competitive tasks (Littman, 1994) as well as mixed tasks (Tesauro, 2003). In this chapter two extensions to Q-Learning are presented. Each of them tries to address a concern or weakness of Q-Learning when dealing with multi-agent or non-stationary tasks. These two algorithms will serve as the basis of novel extensions to large state spaces that will be introduced in the next chapter.

3.1 Repeated Update Q-Learning

3.1.1 Q-Learning Overestimation

In Thrun and Schwartz (1993) an analysis was presented that uncovered issues in the way Q-Learning estimates the action-values. From Equation 2.5 it is known that,

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

If it is assumed that in order to estimate a Q-value, $Q^{target}(s',a)$ its current approximation, $Q^{approx}(s',a)$ equals,

$$Q^{approx}(s',a) = Q^{target}(s',a) + Y^a_{s'} \tag{3.1}$$

Where $Y^a_{s'}$ is the noise, given by a family of random variables with zero mean. This noise factor causes an error at the time of updating $Q(s,a)$. Using Equation 2.5 we assign the result to a random variable Z_s as,

$$Z_s = \gamma(\max_a Q^{approx}(s',a) - \max_a Q^{target}(s',a)) \tag{3.2}$$

Due to the noise, often Z_s will have positive mean such $\mathbb{E}[Z_s] > 0$. These cases lead to an overestimation of the actual $Q(s,a)$. The argument above considers a single update. As the update rule is applied multiple times in order to estimate an action-value this translates into a *systematic overestimation effect*. A second problem in Q-Learning estimation is linked to overestimation and it is referred to as transient bias (Lee and Powell, 2012) or policy bias (Abdallah and Kaisers, 2013). The use

of $\max_a Q(s',a)$ in the estimator favors the selection of larger action-values, because $Q(s,a)$ are biased upwardly this leads to a cascading effect accelerating overestimation for certain state action pairs. For function approximation max induced bias could lead to severe destabilization (Kaelbling et al., 1996).

3.1.2 Addressing Overestimation with RUQL

Repeated Update Q-Learning (RUQL) (Abdallah and Kaisers, 2013, 2016) is an algorithm based on Q-Learning and designed with the intention of addressing its overestimation issues. It follows intuitively from the idea of policy bias. Since in Q-Learning only the action that is selected is updated, then it is said that the effective rate of updating an action-value will depend on the probability of selecting that action. It was described in the previous section, that Q-Learning tends to be upwardly biased. Systematic overestimation self reinforces the tendency to select certain actions by updating them frequently. In non-stationary environments this issue is exacerbated further because previously optimal actions would still be constantly selected even if they are no longer beneficial.

Ideally, if an agent could execute every possible action in parallel but identical environments at each time step, then information about all possible actions could be gathered in order to update every action value simultaneously. From this conjecture, RUQL proposes that an action value must be updated inversely proportional to the probability of the action selected given the policy that is being followed. Thus when an action with low probability is selected, the corresponding action-value is updated more than once. By contrast, if an action with high probability is selected, then the action-value may be updated only once. For example consider two actions A and B, with probability of selecting them $P(A) = 0.8$ and $P(B) = 0.2$ respectively. If action A is selected then its action-value is updated only once, on the other hand if action B is selected then it will be updated five times. Algorithm 5 provides an initial way to formalize this intuition. Where an action-value is updated by $\lfloor \frac{1}{\pi(s,a)} \rfloor$. This implementation however becomes unbounded in computation time as $\pi(s,a) \to 0$.

Algorithm 5 Repeated Update Q-Learning: Intuition

1: Initialize to arbitrary $Q(s,a)$
2: Observe current state s
3: **repeat**
4: Compute policy π using $Q(s,a)$
5: Select an action a according to policy $\pi(s,a)$
6: Execute action a
7: Observe reward r and next state s'
8: **for** $\lfloor \frac{1}{\pi(s,a)} \rfloor$ times **do**
9: $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_a Q(s',a') - Q(s,a)]$
10: **end for**
11: Set $s \leftarrow s'$
12: **until** Termination

Algorithm 6 Repeated Update Q-Learning: Practical Implementation

1: Initialize to arbitrary $Q(s,a)$
2: Observe current state s
3: **repeat**
4: Compute policy π using $Q(s,a)$
5: Select an action a according to policy $\pi(s,a)$
6: Execute action a
7: Observe reward r and next state s'
8: $Q(s,a) \leftarrow [1-\alpha]^{\frac{1}{\pi(s,a)}} Q(s,a) + [1-(1-\alpha)^{\frac{1}{\pi(s,a)}}][r + \gamma \max_a Q(s',a')]$
9: Set $s \leftarrow s'$
10: **until** Termination

In Abdallah and Kaisers (2013, 2016) a closed form expression is derived from the recursive expansion of lines 8-10 from Algorithm 5. The new expression,

$$Q^{t+1}(s,a) = [1-\alpha]^{\frac{1}{\pi(s,a)}} Q^t(s,a) + [1-(1-\alpha)^{\frac{1}{\pi(s,a)}}][r + \gamma \max_a Q^t(s',a')] \tag{3.3}$$

Integrates repeated updates into a single line (Algorithm 6 line 8), and also it removes the need for a floor notation. This provides additional accuracy to obtain the proportion in which action-values should be updated.

In Equation 3.3 it is observed that if an action has a very high chance of being selected then $1/\pi(s,a) \rightarrow 1$ and standard Q-Learning is recovered. On the other hand

when an action is rarely selected then not only the action-value is updated inversely proportional but also the new estimates carry more weight. This addresses the fact that infrequently used actions may contain estimates that are obsolete. This property could lead to more robust behavior in non-stationary environments.

3.1.3 Related Work

A few variants of Q-Learning have been proposed to address overestimation. In Kaisers and Tuyls (2010) an algorithm called Frequency Adjusted Q-Learning (FAQL) is introduced to overcome policy-bias. FAQL and RUQL share similarities as both attempt to update the action-values inversely proportional to the probability of selecting an action. In FAQL the update rule is given by,

$$Q^{t+1}(s,a) = Q^t(s,a) + \frac{1}{\pi(s,a)}\alpha[r + \gamma \max_a Q^t(s',a')]$$

In the same manner as in the preliminary version of RUQL, this implementation is impractical. As $\pi(s,a) \to 0$ the update values become unbounded. The authors offer a practical version by adding a hyperparameter $\beta \in [0,1)$,

$$Q^{t+1}(s,a) = Q^t(s,a) + \min\left(1, \frac{\beta}{\pi(s,a)}\right)\alpha[r + \gamma \max_a Q^t(s',a')]$$

From the expression above, it is observed that once $\pi(s,a)$ goes below β, FAQL will behave as standard Q-Learning. In addition the hyperparameter β reduces the actual learning rate to $\beta\alpha$.

In van Hasselt (2010), Double Q-Learning is proposed to deal with the overestimation produced by the max operator. The algorithm decouples the process of selecting an action from that of evaluating the action. It defines two functions Q^A and Q^B. At each update one of the functions is selected while using the value stored from the other one,

$$Q^A(s,a) \leftarrow Q^A(s,a) + \alpha[r + \gamma Q^B(s', \arg\max_a Q^A(s,a)) - Q^A(s,a)]$$
$$Q^B(s,a) \leftarrow Q^B(s,a) + \alpha[r + \gamma Q^A(s', \arg\max_a Q^B(s,a)) - Q^B(s,a)]$$

Another algorithm tackling overestimation related to the use of max operator is Bias Corrected Q-Learning (Lee and Powell, 2012). A correction term B is introduced to cancel the error in the estimation. The term is derived from the bounds of the bias of the system, leading to an update of the form,

$$Q^{t+1}(s,a) \leftarrow Q^t(s,a) + \alpha[r + \gamma \max_a Q^t(s',a') - Q^t(s,a) - B^t(s,a)]$$

3.2 Loosely Coupled Learning

3.2.1 Overview

We now consider dealing with non-stationary environments from another perspective. Instead of addressing bias or overestimation in Q-Learning as the algorithms presented in the previous section, the following algorithm (Yu et al., 2015) makes explicit considerations about multiple agents. In cooperative distributed environments, agents will find themselves with the necessity to coordinate their actions. Depending on the region or the state of the environment some actions will require a high degree of coordination. In contrast, an agent might find that in other situations coordination with other agents does not improve in any way the decision making process. For this reason, some research has focused on developing more efficient coordinated learning paradigms in which degrees of independence can be exploited by defining cooperation levels (Ghavamzadeh et al., 2006). Given that the state and the action space grows with every agent in an environment, determining when an agent can act independently allows to decompose large distributed problems into smaller decision making processes. In Yu et al. (2015) an algorithm is proposed to determine the degree of independence of an agent. This measure of independence is also dynamically adapted and learned by the agent. In this manner an agent only coordinates or shares information only when it is necessary.

3.2.1.1 Agent independence

An independence degree $\xi_i^k \in [0,1]$ for agent i in state s_i^k determines the probability of an agent carrying on an action independently. The closer ξ_i^k is to the upper bound, the more certainty there is for an agent to act based on its individual information regardless of the presence of other agents.

3.2.1.2 Determining Independence Degree ξ_i^k

The beliefs of an agent to act independently at a given state are adjusted in relation to the negative outcomes it receives. At every state where a negative reward is received, the extent of responsibility of a previous state is determined. A Gaussian-like diffusion function,

$$f^r_{s*}(s) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\zeta_{\langle s,s^*\rangle}2} \tag{3.4}$$

Measures the contribution of preceding state s after receiving a reward at state s^*, where $\zeta_{\langle s,s^*\rangle}$ quantifies the similarity between states. A large value in $f^r_{s*}(s)$ is obtained when the level of similarity between the states is high which indicates a substantial involvement of state s in causing a negative reward. However even if there is a level of similarity between states, it becomes important to recognize what states belong to the state trajectory leading to a negative reward. Through eligibility traces, credit can be assigned to the states involved in a negative reward.

$$\varepsilon^k_i(t+1) = \begin{cases} \gamma^\lambda \varepsilon^k_i(t) + 1 & \text{if } s^k_i \in S^c \\ \gamma^\lambda \varepsilon^k_i(t) & \text{otherwise} \end{cases} \tag{3.5}$$

Here $\varepsilon^k_i(t)$ is the eligibility trace value of agent i in state s^k_i, $\gamma \in [0,1)$ is the discount rate, $\lambda \in [0,1]$ is a decay parameter and S^c is a state trajectory, indicating a series of states involved in a negative reward. If state s^k_i is found in the trajectory, its $\varepsilon^k_i(t)$ increases implying its involvement in an event.

The outcomes from the diffusion function and the eligibility trace are then combined to obtain a value ψ^k_i indicating the necessity for cooperation,

$$\psi^k_i(t+1) = \psi^k_i(t) + \varepsilon^k_i(t) f^r_{s*}(s^k_i) \tag{3.6}$$

ψ^k_i is initialized $\psi^k_i(0) = 0$. A larger value of ψ^k_i corresponds to a more considerable need for cooperating. That is, ψ^k_i is inversely related to independence degree ξ^k_i. Thus ψ^k_i is mapped and bounded to ξ^k_i with a normalization function,

$$\xi^k_i(t+1) = G(\psi^k_i(t+1)) \tag{3.7}$$

Where the larger the value of ψ^k_i the lower and closer to zero, the independence degree ξ^k_i is. The process of adjusting the independence degrees for each agent i at each state k can be summarized in Algorithm 7.

For illustration purposes, Algorithm 7 uses a normalization function $G(.)$ where a tracking variable max_{temp} saves the largest observation of ψ^k_i (Lines 11-13). This variable is later used to scale them in order to be used to obtain an independence degree ξ^k_i (Line 15-17). Similarity between the states is calculated using Euclidean distance (Line 9). Depending on the particular problem, similarity and the normalization function can be modified as required.

Algorithm 7 Independence Degree Adjustment

if negative reward in $s_i(t)$ **then**
  $max_{temp} \leftarrow 0$
  for $s_i^k \in S_i$ **do**
    if $s_i^k \in S_i^c$ **then**
      $\varepsilon_i^k(t) = \gamma^{\lambda} \varepsilon_i^k(t-1) + 1$
    else
      $\varepsilon_i^k(t) = \gamma^{\lambda} \varepsilon_i^k(t-1)$
    end if
    $f_{s*}^r(s_i^k) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}[(x_k - x_u)^2 + (y_k - y_u)^2]}$
    $\psi_i^k(t) = \psi_i^k(t-1) + \varepsilon_i^k(t) f_{s*}^r(s_i^k)$
    if $\psi_i^k(t) \geq max_{temp}$ **then**
      $max_{temp} = \psi_i^k(t)$
    end if
  end for
  for $s_i^k \in S_i$ **do**
    $\xi_i^k(t) = 1 - \frac{\psi_i^k(t)}{max_{temp}}$
  end for
end if

3.2.1.3 Coordinated Learning

Once the elements to determine the need for coordinated action have been established, it is now analyzed how it is included into the learning process. Supported by the Dec-MDP framework introduced in Section 2.1.2.1, the case of full joint observability is considered when combining the local observations of every agent. For simplicity and for future empirical testing, only two agents i and j are considered. Thus the joint action is given by $a =< a_i, a_j >$ and the joint state by a factored representation $S = S_i \times S_j$ or $S = S_0 \times S_i \times S_j$ depending on the existence of an agent independent component S_0. With the inclusion of an independence degree then the problem can be decomposed into sub-problems where an MDP is solved when an agent is acting individually and another MDP with a fully observable joint state when acting in coordination with the other agent. As it has been the case, reinforcement learning algorithms can be used to find policies for these MDPs.

Two Q-functions are defined for calculating action values for an agent i, Q_i when acting individually and Q_c when acting in coordination with the other agent. Using the

Algorithm 8 Coordinated Learning for agent i

Initialize $Q_i(s_i, a_i)$ and $Q_c(js_i, a_i)$
Initialize $\xi_i^k(t) = 1$, $\varepsilon_i^k(t) = 0$ and $S^c \leftarrow \emptyset$
repeat
  Generate random number $\tau \sim \mathcal{U}(0,1)$
  if $\xi_i^k \leq \tau$ **then**
    if agent j is observed **then**
      Set $perceptionFlag = True$
      Set joint state $js_i \leftarrow \langle s_i, s_j \rangle$ to try to coordinate with agent j
      Select an action a_i according to policy π w.r.t. $Q_c(js_i, a_i)$
    else
      Select an action a_i according to policy π w.r.t. $Q_i(s_i, a_i)$
    end if
  else
    Select an action a_i according to policy π w.r.t. $Q_i(s_i, a_i)$
  end if
  Include state s_i in trajectory states S^c
  Execute action a_i
  Observe reward r_i and next state s_i'
  if $(\xi_i^k \leq \tau)$ AND $(perceptionFlag == True)$ **then**
    $Q_c(js_i, a_i) \leftarrow Q_c(js_i, a_i) + \alpha[r_i + \gamma \max_{a_i} Q_i(s_i', a_i') - Q_c(js_i, a_i)]$
  else
    $Q_i(s_i, a_i) \leftarrow Q_i(s_i, a_i) + \alpha[r_i + \gamma \max_{a_i} Q_i(s_i', a_i') - Q_i(s_i, a_i)]$
  end if
  Adjust ξ_i^k using Algorithm 7
  Set $s_i \leftarrow s_i'$
until Termination

Q-Learning update rule from equation 2.6,

$$Q_i(s_i, a_i) \leftarrow Q_i(s_i, a_i) + \alpha[r_i + \gamma \max_{a_i} Q_i(s_i', a_i') - Q_i(s_i, a_i)] \tag{3.8}$$

Which estimates the action-value for an agent i acting independently. Meanwhile when acting in a coordinated manner, a joint state js_i is considered. The joint state can be understood as the information shared between the agents about their observations of the environment. Using again the Q-learning update rule,

$$Q_c(js_i, a_i) \leftarrow Q_c(js_i, a_i) + \alpha[r_i + \gamma \max_{a_i} Q_i(s'_i, a'_i) - Q_c(js_i, a_i)] \tag{3.9}$$

It has to be emphasized that it is Q_c the Q-function used to estimate in case of a coordinated action. However in its max operator, the estimate uses the individual Q_i to account for the value of future actions. From Algorithm, 8 it is observed that an action that considers shared information could only be taken when the other agent j is in a situation where it is observed, or is in a position to transmit its information (Line 6-9). Thus when agent j is unavailable, agent i will act independently regardless of its need to coordinate (Line 10-12).

3.2.2 Related Work

Previous research has tried to address the issue of equilibrating independence and coordination. In Roth et al. (2007), using factored Dec-MDP as in (Yu et al., 2015) they provide a different approach by building and computing tree structured policies. They describe a mechanism to transform factored joint policies into factored individual policies. Another influential method in multi-agent learning, involves using graph structures and solving the MDP through a message passing scheme (Guestrin et al., 2002). The structure of the graph describes the relevant variables that an agent should take into account in order to maximize a joint action. Kok and Vlassis (2004) presented a similar approach to the one exposed here. In their paper they describe the Sparse Cooperative Q-Learning algorithm, each agent considers an individual and a collective Q-function. However it differs from Yu et al. (2015) in that it relies on a coordination graph to determine in which states the agents must act jointly, and to extract value rules that are added to the global Q-function. In subsequent work, Kok et al. (2005) attempt to learn automatically the structure of the coordination graphs. Initially all agents start acting independently and as they discover states in need of coordination, value rules are added to the coordination graph. Coordination requirements are calculated based on statistics about the actions selected by the other agents. These methods based on coordinated graphs assume a predefined interaction structure. In other relevant work by De Hauwere et al. (2009), they propose a more general approach. The decision making process is decoupled into two layers. A first layer uses generalized linear automata to learn associations between rewards and state information. Then a second layer decides to use standard Q-learning or a multi-agent algorithm. They provide an alternative approach in De Hauwere et al. (2010), in which the agent builds a representation of

the task by computing statistics about the change in rewards in visited states. However there is the assumption that an agent has already learned an individual optimal policy before it can start learning to coordinate with other agents.

Chapter 4

Extending to Multi-Agent Deep Reinforcement Learning

In the previous chapter, two algorithms designed to deal with non-stationarity were reviewed. Repeated Update Q-Learning (RUQL) tackles policy bias observed in Q-Learning. The second algorithm, which will be now referred to as Loosely Coupled Q-Learning (LCQL), attempts to balance independent decision making with coordinated action for multi-agent environments. In this chapter two variants inspired by these algorithms are presented. The objective is to extend and generalize to multi-agent domains with large state spaces using deep neural networks.

4.1 Deep Repeated Update Q-Learning

The update rule of RUQL is given by Equation 3.3 as,

$$Q^{t+1}(s,a) = [1-\alpha]^{\frac{1}{\pi(s,a)}} Q(s,a) + [1-(1-\alpha)^{\frac{1}{\pi(s,a)}}][r + \gamma \max_a Q(s',a')]$$

Setting $z_{\pi(s,a)} = 1-(1-\alpha)^{\frac{1}{\pi(s,a)}}$ and $\omega = 1 - z_{\pi(s,a)}$ the equation can be expressed as (Appendix A),

$$Q^{t+1}(s,a) = Q(s,a) + z_{\pi(s,a)}[r + \gamma \max_a Q(s',a') - Q(s,a)] \tag{4.1}$$

For function approximation, it has been established that a lookup table with a correspondence of 1 to 1 for each state action pair is substituted by an approximation of the action-value. In this case,

$$\hat{Q}(s,a;\theta) = g\left(\sum_{j}^{n} \theta^{(j)}\phi^{(j)}(s,a)\right)$$

θ is a vector of parameters or weights for the neural network, ϕ is the input and constitutes any form of pre-processing or transformation applied to the state, and the function $g(.)$ is a nonlinearity. For updating the parameters θ, an expression of the following form must be obtained,

$$\theta_{i+1} \leftarrow \theta_i + \alpha\nabla_{\theta_i} L(\theta_i)$$

Where i is the current iteration, α is a learning rate and $L(.)$ is a loss function. From Equation 4.1 the following loss function can be defined,

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[r + \gamma \max_a Q(s',a';\theta_i^-) - Q(s,a;\theta_i)]^2$$

State s, action a, reward r and next state s' are uniformly sampled from replay memory D that stores transitions. θ_i^- contains the parameters of a previous iteration thus we can set $y_i = \mathbb{E}_{(a,r,s')\sim U(D)}[r + \gamma \max_a Q(s',a';\theta_i^-)$ as the target. Differentiating the above loss function with respect to the parameters,

$$\alpha\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s',z_{\pi(s,a)})\sim U(D)}[\mathbf{z}_i(y_i - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)] \tag{4.2}$$

In this expression, the learning rate has been included to clarify its use in RUQL. Standard gradient descent considers a global learning rate α for all parameters. However, for RUQL we require more granularity. A vector $\mathbf{z}_i$ contains the effective learning rates or step sizes $z_{\pi(s,a)} = 1 - (1-\alpha)^{\frac{1}{\pi(s,a)}}$. Each of these is associated to an individual element of a minibatch. The reason is that $z_{\pi(s,a)}$ is calculated in relation to the action selected when following a determined policy at a given time step. Thus $\pi_\theta(s,a) = P[a|s;\theta]$ which depends on the exploration strategy used and its hyperparameter values at the time of selecting an action.

This process can be observed in Algorithm 9 where after selecting an action the individual effective learning rate is computed (Line 8). This information is then stored together with the rest of the agent experience in the replay memory D (Line 12). Once the network is about to update its parameters the transitions are sampled from the stored memory (Line 13). In the particular case of the effective learning rates $z_{\pi(s,a)}$, they are used to populate the vector $\mathbf{z}_i$ which is then used to determine how to scale the gradient (Lines 15-17).

Algorithm 9 Deep Repeated Update Q-Learning

Initialize replay memory D to hold N transitions
Initialize action-value function Q with random weights θ
for episode=1...M **do**
  Initialize a sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for episode=1...T **do**
    Select an action a_t according to policy π
    Compute effective learning rate $z_{\pi(s_t,a_t)}$
    Execute action a_t
    Observe reward r_t and next frame x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1}, z_{\pi(s,a)_t})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1}, z_{\pi(s,a)_{j+1}})$ from D
    Set $y_j = \begin{cases} r_j & \text{if terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Populate $\mathbf{z}_i$ with sampled $z_{\pi(s,a)}$
    Perform gradient descent on $(y_j - Q(\phi_j, a_j; \theta))^2$ wrt θ
    Update $\theta_{i+1} \leftarrow \theta_i + \mathbf{z}_i \nabla_{\theta_i} L(\theta_i)$
    Every C steps set $\hat{Q} = Q$ ▷ If using an extra target network
  end for
end for

4.2 Deep Loosely Coupled Q-Learning

4.2.1 Determining a Single Independence Degree ξ_k

To decide when an agent must coordinate or act independently, LCQL defined an independence degree ξ, which itself relied on calculating an eligibility trace ε and a coordination measure ψ. All these variables assumed a small state space where every possible state is known or can be stored and retrieved. In practice this is unfeasible. This section presents a *Deep Loosely Coupled Q-Network (DLCQN)*. The issue of calculating these measures is tackled by defining a single measure of independence for all states. As in the case of the original formulation of LCQL, the independence degree is automatically adjusted when a negative outcome is observed.

A diffusion function is defined from the state s_k^* where a negative outcome is received by agent k. Instead of determining the responsibility of its own previous states, the

Algorithm 10 Independence Degree Adjustment for agent k (DLCQL)

if negative reward in $s_i(t)$ **then**
  $idSteps \leftarrow idSteps + 1$
  if agent j was observed **then**
    $f^r_{s^*_k}(s_j) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}[(x_k - x_j)^2 + (y_k - y_j)^2]}$
    $\varepsilon_k(t) = \gamma^\lambda \varepsilon_k(t-1) + 1$
    $\psi_k(t) = \psi_k(t-1) + \varepsilon_k(t) f^r_{s*}(s_k)$
  else
    $\varepsilon_k(t) = \gamma^\lambda \varepsilon_k(t-1)$
    $\psi_k(t) = \gamma^\lambda \psi_k(t-1)$
  end if
  if $\psi_k(t) \geq max_{temp}$ **then**
    $max_{temp} = \psi_k(t)$
  end if
  if $idSteps\%U == 0$ **then** ▷ Every U steps ξ is updated
    $\xi_k(t) = 1 - \frac{\psi_k(t)}{max_{temp}}$
    $max_{temp} = 0$
  end if
end if

agent calculates the influence of the state information shared by the other agent j. First the similarity $\zeta_{\langle s_j, s^*_k \rangle^2}$ between states is computed, then a diffusion function can be calculated with,

$$f^r_{s^*_k}(s_j) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\zeta_{\langle s_j, s^*_k \rangle^2}} \tag{4.3}$$

Similarly, the eligibility trace ε_k is constrained to only one per agent instead of one per state. The eligibility trace remains unchanged however it serves a distinct role from LCQL. Originally the eligibility trace was intended to assess the influence of previous states leading to a negative outcome. In the case of a DLCQN, the factor ε_k, helps to establish a measure or association between receiving a negative reward and observing another agent.

$$\varepsilon_k(t+1) = \begin{cases} \gamma^\lambda \varepsilon_k(t) + 1 & \text{if agent } j \text{ was observed} \\ \gamma^\lambda \varepsilon_k(t) & \text{otherwise} \end{cases} \tag{4.4}$$

$\gamma \in [0,1)$ is the discount rate and $\lambda \in [0,1]$ is a decay parameter. In the same manner ψ_k that gauges the necessity for coordination, increases when there is a correlation

between receiving a negative reward and observing another agent and decreases when it is received and no agent has been observed.

$$\psi_k(t+1) = \begin{cases} \psi_k(t) + \varepsilon_k(t) f^r_{s^*_k}(s_j) & \text{if agent } j \text{ was observed} \\ \gamma^\lambda \psi_k(t) & \text{otherwise} \end{cases} \tag{4.5}$$

Algorithm 10 describes the steps necessary to update the independence degree ξ_k for agent k. As was the case in LCQL, a larger ψ_k will correspond to a lower independence degree. However instead of normalizing ψ_k with respect to every state available for an agent, it is normalized against the highest psi_k calculated during the U previous observations of negative rewards (Line 11-17).

4.2.2 Approximating the Local and Global Q-functions

The same basic principles applied to RUQL regarding function approximation using neural networks are then followed for LCQL. In this case however, there are two Q-functions: (1) $\hat{Q}_k$ approximates action-values when agent k acts independently and (2) $\hat{Q}_c$ which will be defined as a global Q-function when agents act coordinately. Thus,

$$\hat{Q}_k(s,a;\theta^k) = g\left(\sum_j^n \theta^{(j)}\phi^{(j)}(s,a)\right)$$
$$\hat{Q}_c(js,a;\theta^c) = g\left(\sum_j^n \theta^{(j)}\phi^{(j)}(js,a)\right)$$

Since every individual function represents a separate network, each of them uses their own set of parameters θ. In addition, Q_c computes its value using an extended joint state js that incorporates the state information of each agent. A target can be set as $y_i^k = \mathbb{E}_{(a,r,s')\sim U(D)}[r + \gamma \max_a Q(s',a';\theta_i^{k-})$ for agent k. This same target y_i^k is also used when agent requires to coordinate and receives information from another agent. Based on the original equations 3.8 and 3.9 used to update the Q-values, these are adapted to obtain the following loss functions,

$$L_i^k(\theta_i^k) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[(y_i^k - Q_k(s,a;\theta_i^k))]^2 \tag{4.6}$$
$$L_i^c(\theta_i^c) = \mathbb{E}_{(js,a,r,s')\sim U(D)}[(y_i^k - Q_c(js,a;\theta_i^c))]^2 \tag{4.7}$$

Algorithm 11 Deep Loosely Coupled Q-Learning for agent k

Initialize replay memory D to hold N transitions
Initialize Q_k to random θ^i and Q_c to random θ^c
Initialize $\xi_k(t) = 1$, $\varepsilon_k(t) = 0$
Initialize $max_{temp} = 0$, $idSteps = 0$
for episode=1...M **do**
    Initialize a sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    for episode=1...T **do**
        Set $perceptionFlag = False$
        Generate random number $\tau \sim \mathcal{U}(0,1)$
        if $(\xi_k \leq \tau)$ AND (agent j is observed) **then**
            Set $perceptionFlag = True$
            Set joint state $js_k \leftarrow \langle s_k, s_j \rangle$ to try to coordinate with agent j
            Preprocess sequence $\phi(js_k)$
            Select an action a_k according to policy π w.r.t. $Q_c(\phi(js_k), a_k)$
        else
            Select an action a_k according to policy π w.r.t. $Q_k(\phi(s_k), a_k)$
        end if
        Execute action a_k
        Observe reward r_t and next frame x_{t+1}
        Set $s_{k,t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        Set $y_j^k = \begin{cases} r_j & \text{if terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{k-}) & \text{otherwise} \end{cases}$
        if $(\xi_k \leq \tau)$ AND $(perceptionFlag == True)$ **then**
            Build joint representation $\phi(js_k)$ from sampled $\phi(s_k)$
            Perform gradient descent on $(y_j^k - Q_c(\phi(js_k)_j, a_j; \theta))^2$ wrt θ
            Update $\theta_{i+1}^c \leftarrow \theta_i^c + \nabla_{\theta_i^c} L^c(\theta_i^c)$
        else
            Perform gradient descent on $(y_j^k - Q_k(\phi(s_k)_j, a_j; \theta))^2$ wrt θ
            Update $\theta_{i+1}^k \leftarrow \theta_i^k + \nabla_{\theta_i^k} L^k(\theta_i^k)$
        end if
        Every C steps set $\hat{Q} = Q$ ▷ If using an extra target network
        Adjust ξ_k using Algorithm 10
        Set $s_k \leftarrow s_k'$
    end for
end for

Then similarly to Equation 4.2, the gradients for each loss function are given by,

$$\nabla_{\theta_i^k} L_i^k(\theta_i^k) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[(y_i^k - Q_k(s,a;\theta_i^k))\nabla_{\theta_i^k} Q_k(s,a;\theta_i^k)] \tag{4.8}$$

$$\nabla_{\theta_i^c} L_i^c(\theta_i^c) = \mathbb{E}_{(js,a,r,s')\sim U(D)}[(y_i^k - Q_c(js,a;\theta_i^c))\nabla_{\theta_i^c} Q(js,a;\theta_i^c)] \tag{4.9}$$

Where c indicates the elements belonging to a coordinated action and k to those of a corresponding agent k.

4.3 Prior Research

Although Deep Reinforcement Learning is a new area of research, interest in multi-agent environments has grown quickly. The earliest precedent on multi-agent deep reinforcement learning (MADRL) is from Tampuu et al. (2015). Their work is also the most closely related to our own. They considered two agents in *Pong*, testing various rewarding schemes. Each agent was controlled by its own DQN but no further modification was done. Thus their work was a direct extension to the game mode rather than an extension to the DQN architecture. In Egorov (2016), MADRL was explored in the context of a pursuit evasion task. Image-like representation were created from the grid in which the agents were located, and then split into different channels. Each of them corresponding to a mapping either to obstacles, allies, opponents or the current location of the agent. Although the architecture diverged from the original DQN by using Residual networks, the mechanisms behind the RL component remained the same. In addition, each agent was trained and learned individually as opposed to in parallel.

Other recent studies have explicitly integrated communication as an essential component to address partial observability between the agents. In these models communication is not predefined but learned (Sukhbaatar et al., 2016). Foerster et al. (2016a) build on the single agent Deep Recurrent Q-Network (Hausknecht and Stone, 2015) and extend it to a multi-agent domain where network parameters are shared. Foerster et al. (2016b) added to this approach by passing gradients through the agents. These architectures fundamentally differ from DLCQN and other research dealing with POMDPs (Egorov, 2015) in that they use or partially depend on recurrent neural networks to process non-Markovian states.

In the same manner that DRUQN generalizes RUQL from tabular lookup tables to large state spaces using deep convolutional neural networks, Double Q-Learning was also recently applied to adapt the original DQN parameter update rule (van Hasselt

et al., 2015). The resulting network, Double DQN, surpassed the performance of DQN in the Atari 2600 games.

Chapter 5

Methodology

5.1 Experimental Setup

The performance of the models is tested in the two-player game *Pong*. Recent research in reinforcement learning has used extensively the Arcade Learning Environment (Bellemare et al., 2012) as a testbed for Atari 2600 games. However the framework does not currently support multi-player control. For these experiments an open source version of Pong, available from the Pygame repositories is used as a substitution. Using this alternative also permitted customizing the game to explore different modalities. Pong is a table tennis game in which each player controls a paddle at opposing sides of the screen in order to hit a ball. In the original form of the game a point is awarded every time the opponent fails to hit the ball back. There are three possible actions for each paddle: (1) to move up, (2) to move down and (3) not to move. In a similar way to the experiments in Mnih et al. (2013, 2015) a game finishes when one of the two players reaches 20 points. Although other multi-player Atari games are available, Pong was selected because it allows a simultaneous two player mode. This is necessary to test the application of the algorithms in non-stationary environments and in situations that may require certain level of coordination. In addition, the reward structure of the game can be intuitively modified to test different goals. Pong was also one of the games that DQN learned to play efficiently thus it allows to establish initial comparisons between the different approaches presented in this work. In the rest of this section, the three type of testing scenarios designed to evaluate the performance of the algorithms are described.

5.1.1 1-Player Control

The first experimental modality studied is when only one of the players is controlled by a network and the other by the game AI. Three different instances for this task are considered:

- **DQN**: The test serves for baseline results, to try to establish the performance of a network based on Mnih et al. (2013, 2015) under these particular experimental conditions.
- **DRUQN**: In a similar way to DQN, a single network controls a player against the game AI.
- **DLCQN**: In contrast to the other two algorithms, DLCQN explicitly assumes the existence of multiple players. In this case there are two networks that are trained simultaneously. A local network that corresponds to the network controlled player and a second network that holds joint state information. This second network is only used by the network-controlled player when the game AI is observed.

The default competitive game mode is used for this set of experiments. Table 5.1 shows the rewards obtained by the network controlled player when it scores and when the game AI scores.

	Network scores	Game AI scores
Network reward	+1	-1

Table 5.1 Reward structure in Pong. Competitive game mode with a single network controlled player.

5.1.2 2-Player Control

Although for the previous modality the network is learning to interact in a multi-agent environment. The agent provided by the game AI has already a well defined behavior. We now want to test the performance of the algorithms in a truly non-stationary environment where both agents are continuously modifying their behavior. Thus we now consider a cooperative task, where each agent is controlled by its own neural network. In this modality, the objective of the task is to keep the ball bouncing between the paddles. Therefore the agents must try to learn policies that allow them to coordinate with each other. The reward scheme is changed to reflect the objectives

of the task (Table 5.2). Every time the ball passes through one of the players both are penalized regardless of who scored. Hitting the ball does not warrant a positive reward, consequently the agents can only minimize their losses in conjunction with each other.

	Net. 1 scores	Net. 2 scores	Net. 1 hits ball	Net. 2 hits ball
Network 1 reward	-1	-1	0	0
Network 2 reward	-1	-1	0	0

Table 5.2 Reward structure in Pong in a cooperative coordination game mode. Both players are controlled by their own network.

Three experimental instances are again considered for this modality:

- **DQN**: Two networks of the same type (DQN) are trained simultaneously. Each of them corresponds to an individual agent.
- **DRUQN**: Two networks also of the same type (DRUQN) each controlling one agent are trained simultaneously.
- **DLCQN**: For this case three DLCQN networks are trained in parallel. Two individual networks, each corresponding to a player, and used only when the agents are acting based on their own independent observations. The third one, is a joint state network shared between both agents. This network attempts to emulate collective decision making when the agents are in a position in which they can transmit their information. Thus this network is used by any of the agents when it requires to act coordinately by integrating its observations with those coming from the other agent.

5.1.3 2-Player Control Mixed

The third set of experiments is intended to establish a direct comparison between different learners in the same environment. For this purpose a standard competitive game mode is considered as described in Table 5.3. The following three instances are tested:

- **DQN vs DRUQN**: The two networks are trained simultaneously, each controlling an agent.

- **DQN vs DLCQN**: Three networks are trained in parallel. An agent is controlled by the DQN. Meanwhile, the other agent has two networks at its disposal for decision making (DLCQN). The first of those is used when the agent decides to act based on its own observations. The second network is used when information originating from the other player is available and the agent decides to use that information to determine how to act.

- **DRUQN vs DLCQN**: In the same manner as above. One of the agents is controlled by a DRUQN. The other agent can either act using only its own information or if available, use joint state information (DLCQN).

	Network 1 scores	Network 2 scores
Network 1 reward	+1	-1
Network 2 reward	-1	+1

Table 5.3 Reward structure in Pong in a competitive game mode. Both players are controlled by their own network.

5.2 Architecture

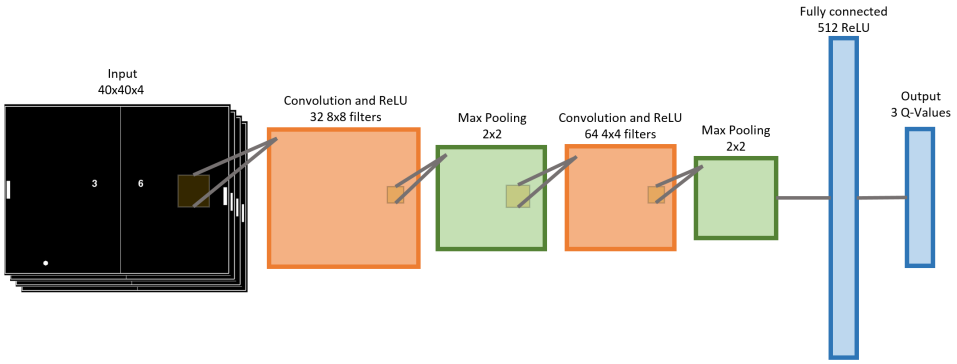

Figure 5.1 Network architecture for DRUQN and DLCQN (local). The input consists on the four most recent frames. The input is then transformed by two hidden convolutional+max-pool layers. Then followed by another hidden fully connected hidden layer. The output layer contains the Q-value estimated for each of the three possible actions.

In essence, the networks receive as input the raw pixels of the screen and output the Q-values of the actions available to an agent. The most recent four frames from the game are first preprocessed by grayscaling and downsampling them. Although the

original architecture in Mnih et al. (2013, 2015) receives frames of 84×84, for our networks we consider frames of 40×40 for memory efficiency purposes. Then the frames are stacked together to form an input of $40 \times 40 \times 4$ and fed into the network. The architecture as shown in Figure 5.1, consists of a first hidden convolutional layer with 32 filters of shape 8×8 with stride of 4×4. The layer applies ReLU and then goes through max-pooling of 2×2. The second hidden convolutional layer has 64 filters of 4×4 with stride 2×2 applying ReLU nonlinearities and then max-pooling of 2×2. Then a final hidden layer of 512 fully connected ReLU neurons. The output layer is a fully connected linear layer that outputs the predicted Q-value for each of the three actions: (1) move up, (2) move down and (3) stand still.

5.2.1 DLCQN Joint Network

For DLCQL, in addition to the agent's own local network (which follows the specification described above) a global network may be used. This network is shared between both agents and contains the joint state information from combining the observations from each of them when they are in each other's field of vision. The original algorithm DLCQN is based on by Yu et al. (2015), was originally intended to solve two robot navigation tasks. The extension proposed here should also work on those tasks. In those scenarios the agents combine the observations of the local region in which they are located. However the nature of those tasks is very different when compared to Pong. Thus the criteria used to determine if an agent is *observed* by the other has to be redefined. We consider that an agent is observed when at least part of its paddle is aligned in parallel with the paddle of the other agent. An example of a situation when an agent uses its local network, and when it has the option to use the shared network is depicted in Figure 5.2 left and right respectively.

The joint state will be constituted by eight preprocessed and stacked frames. The first four are the most recent frames observed by the agent. For the rest it is assumed that the other player has information about the four previous frames. Thus the joint state is given by stacking the most recent eight frames. Accordingly the architecture of the joint state network is augmented to accommodate for a larger input of $40 \times 40 \times 8$. The rest of the layers remain the same and as specified in the previous section.

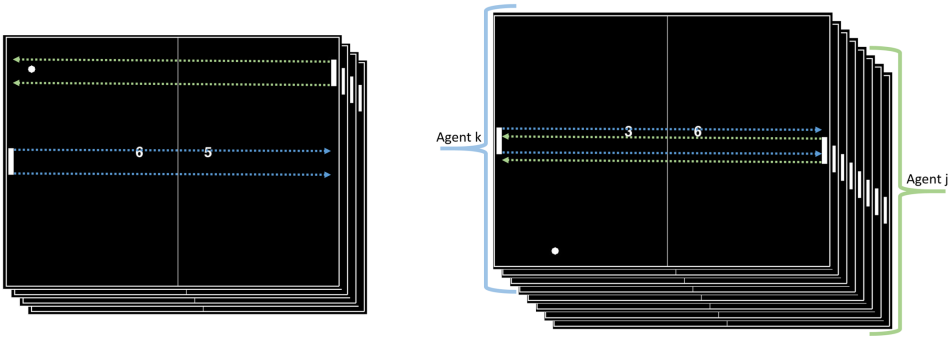

Figure 5.2 Left: The paddle of the agents are not aligned in parallel and it is considered that they are not in each other's field of vision. Only the four most recent frames are sent as input to the agent local network. Right: Here, the paddles are aligned and information about the previous four frames can be shared between the agents. If the agent decides to act based on the shared information, eight frames are passed to the global shared network.

5.2.2 Training

All networks are trained for 102 epochs of 20,000 time steps each for a total of 2 million and 40 thousand frames. Every network is trained using the same parameters and procedures. During the first 50,000 time steps the network starts by executing random actions in order to start populating the replay memory, which holds up to 500,000 transitions. The exploration policy is ϵ-greedy annealing ϵ from 1 to 0.05 during the first 500,000 time steps and remaining at $\epsilon = 0.05$ thereafter. The discount rate is $\gamma = 0.99$ Frame skipping is set to $k = 4$ which indicates that the agent selects an action every kth frame and repeats it. ADAM (Kingma and Ba, 2014) is used for stochastic optimization, with a minibatch of size 200 and a learning rate $\alpha = 1 \times 10^{-6}$. As explained in Section 3.1 and 4.1, for DRUQN the *effective learning rate* is $z_{\pi(s,a)} = 1 - (1-\alpha)^{\frac{1}{\pi(s,a)}}$, where $\pi(a|s) = P[a|s]$. The full list of parameters can be found in Appendix C.

5.2.2.1 Parameters Specific to DLCQN

For training the DLCQN networks we require additional parameters. The initial independence degree $\xi_k(0) = 1$, $\varepsilon_k(0) = 0$, $\psi_k(0) = 0$ and $\lambda = 0.8$ where k is the agent. The agent updates its independence degree after receiving 250 negative rewards.

5.3 Evaluation

At the end of every epoch, the weights are saved and stored in order to evaluate the performance of the networks at that specific point. For each checkpoint, the networks

are evaluated by playing 30 games with a fixed ϵ-greedy exploration policy of $\epsilon = 0.01$. The reward and the scores per game are then averaged. For the cooperative task also the number of bounces per point is calculated using the same methodology described in Tampuu et al. (2015). They are averaged first over a single a match and then over the 30 matches. In the competitive 2-player mixed modality, the mean average reward over a range of epochs is also computed.

However, as has been noted in Mnih et al. (2013, 2015) these metrics tend to be noisy and may give the impression that learning is not occurring. Thus we also follow the approach specified there and in related work. We calculate the average maximal Q-values. This will also give us an indication of whether or not a network is converging. To calculate them, first 500 frames are randomly selected before any training starts. Then similarly to the other metrics, for each checkpoint the 500 frames are passed to the network. The average maximal Q-value is calculated with,

$$\frac{1}{N}\sum_{n=1}^{N}\arg\max_{a} Q(s,a;\theta)$$

where $N = 50$ is the number of frames fed into the network. It has to be mentioned that the same frames are always presented to each network.

Chapter 6

Emprical Evaluation

6.1 1-Player Control

During the first evaluation the performance of the algorithms was very similar. DRUQN and DLCQN agents obtain larger rewards earlier than DQN (Figure 6.1 Top). Although as it was mentioned earlier, averaged rewards tend to be a noisy behavioral descriptor, the three agents learn how to play successfully obtaining full scores routinely.

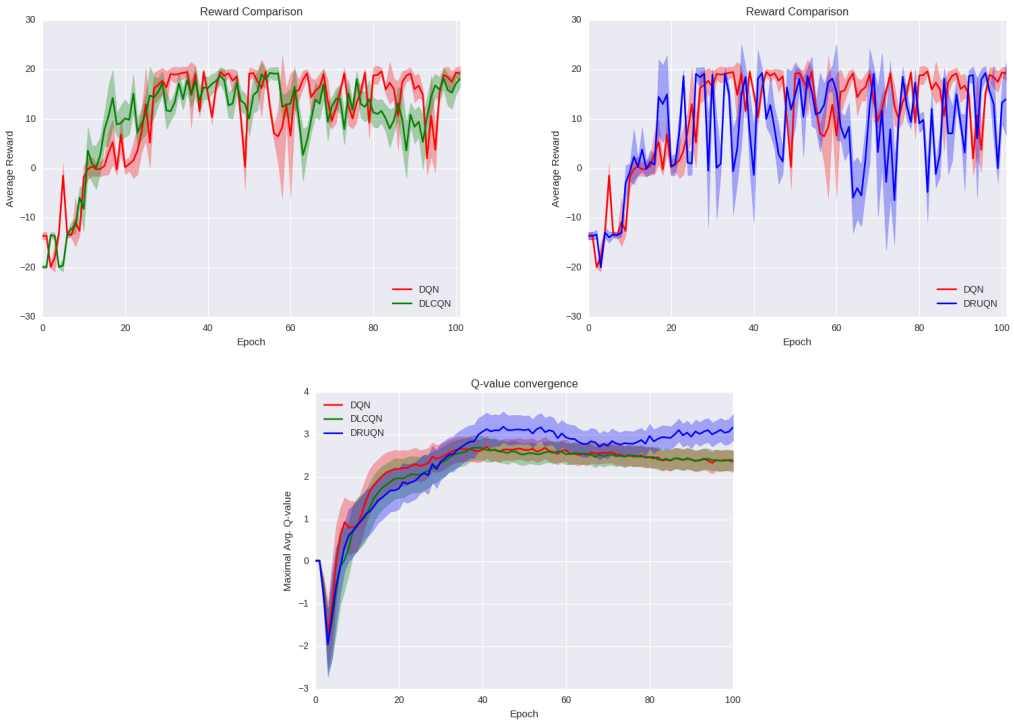

Figure 6.1 Performance comparison between DQN, DRUQN and DLCQN in the 1-Player control task. Top left: Average rewards of DQN and DLCQN. Top right: Average rewards of DQN and DRUQN. Bottom: Average maximal Q-values.

In Figure 6.1 Bottom, we can observe the convergence of the Q-values. The results surprisingly illustrate that DQN and DLCQL using the standard Q-Learning update rule, estimate the values at a lower level than DRUQN. However it is worth to remember that at this stage the environment is stationary. The agents are facing the game AI which has a defined static behavior from the first epoch to the last. Thus amplification effects on the estimation of the values may not be as severe during these instances.

Table 6.1 below, shows the results from each agent in their respective best performing epoch. It can be appreciated that the difference in performance is minimal. The three agents were capable of winning their 30 matches with a score of 20 and conceding less than one point on average. The only significant difference between the agents is that DRUQN and DCLQN achieve their best performance sooner than DQN.

	Mean Reward	Mean Score	Mean Game AI Score	Epoch
DQN	**19.63**±0.75	20	**0.36**±0.75	83
DRUQN	19.3±0.86	20	0.7±0.86	**43**
DLCQN	19.33±1.07	20	0.66±1.07	56

Table 6.1 DQN, DRUQN and DLCQN performance comparison in the 1-Player control task.

6.2 2-Player Control

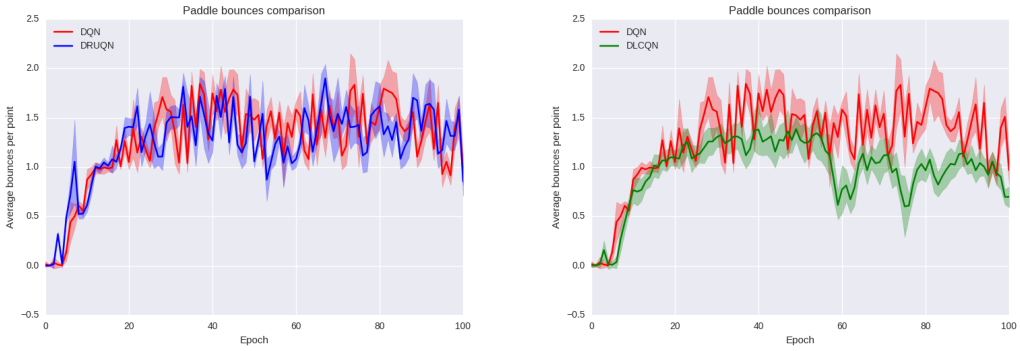

Figure 6.2 Performance comparison between DQN, DRUQN and DLCQN in the 2-Player control cooperative task. Left: Average paddle bouncing for DQN and DRUQN. Right: Average paddle bouncing between DQN and DLCQN.

For the cooperative task, Figure 6.2 depicts the average number of bounces per point achieved when the agents are trying to coordinate with each other. The graphs followed

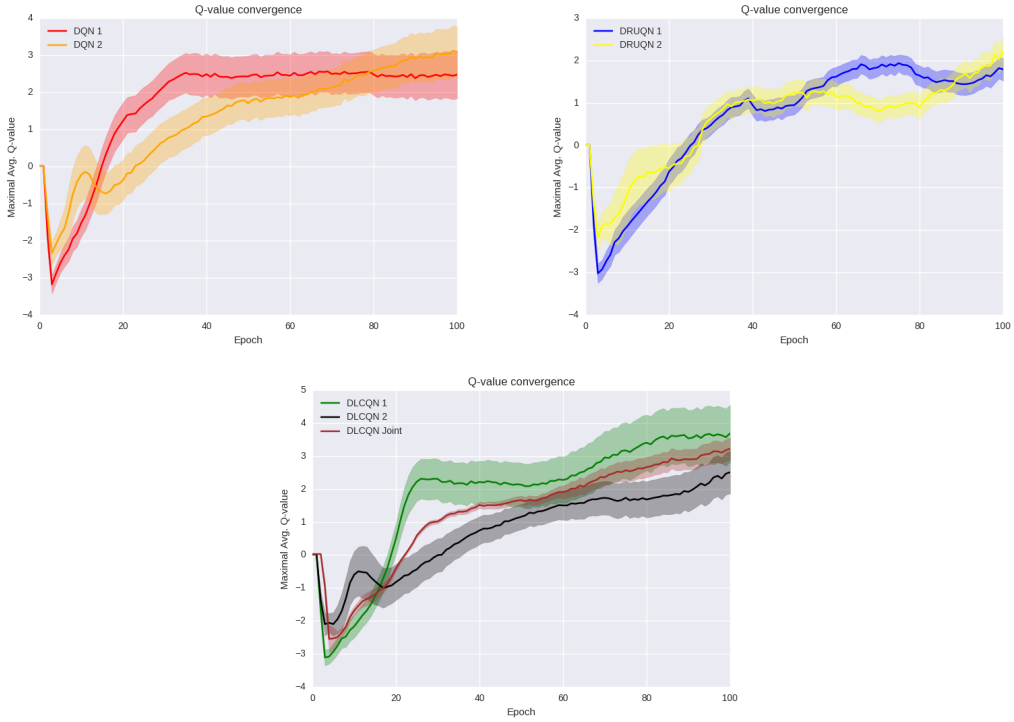

Figure 6.3 Convergence in the 2-Player control cooperative task. Top left: DQN networks. Top right: DRUQN networks. Bottom: DLCQN shared and local networks.

closely the evolution of the joint reward function. DQN and DRUQN agents roughly exhibit the same performance. In the case of DLCQN agents, they start learning at a similar rate to DQN. However the performance of these agents plateaus prematurely. It can even be noticed that the agents tend to a decrease their performance in later epochs. Table 6.2 offers a comparison between each group. The results confirm the observations from the graph. At their peak, the DRUQN agents are able to coordinate the most by maximizing the joint rewards. DLCQN agents on the other hand achieve inferior returns compared to the other two groups, even when considering their best performing epoch.

	Mean Reward	Mean Bouncing	Epoch
DQN	-1.9±1.97	1.84±0.15	**38**
DRUQN	**-1.3**±1.96	**1.89**±0.14	68
DLCQN	-8.9±4.96	1.37±0.25	41

Table 6.2 DQN, DRUQN and DLCQN performance comparison in the 2-Player control cooperative task.

From observing Figure 6.3 it is appreciated that DQN and DRUQN attain certain convergence. For DLCQN only one of the local network reaches an initial degree of stability. The joint and the second network does not fully stabilize as they keep

increasing their estimates. The lack of convergence from these two networks is probably what ultimately affects the first one (green). The volatility in the estimates may explain the declining performance observed for the DLCQN agents. Among the three groups, only in DRUQN, both networks tend towards convergence simultaneously. Unlike in the previous experiments with only one network, this scenario also serves to verify the impact of non-stationarity in the estimates. In fact DQN and DLCQN overestimate in relation to DRUQN. Their values rise above 2 and sometimes beyond 3, whereas DRUQN networks estimate below 2. Nonetheless for this example overestimation does not necessarily have an adverse effect in the learning of the task. This is exemplified by DQN, which exhibits comparable performance to DRUQN.

6.3 2-Player Control Mixed

6.3.1 DQN vs DLCQN

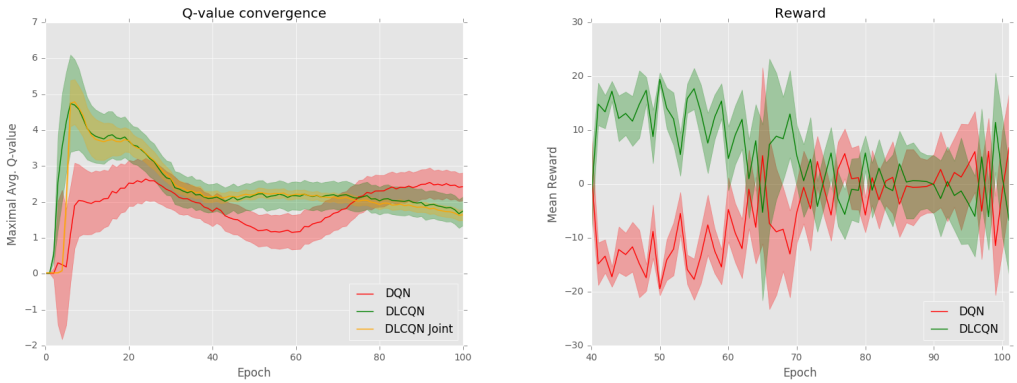

Figure 6.4 DQN vs DLCQN competitive mode. Left: Q-value convergence. Right: Reward when competing against each other.

	Mean Reward	Mean Score	Mean Score Opp.	Epoch
DQN (*vs DLCQN*)	6.73±10.13	17.86±4.57	11.13±6.63	102
DLCQN (*vs DQN*)	**19.43**±1.25	**20**	**0.56**±1.25	**51**

Table 6.3 DQN vs DLCQN competitive mode. First row shows the results when DQN performed the best against DLCQN. The second row is the best performing epoch of DLCQN against DQN.

We first compare a DQN agent competing against another using a local and a joint state DLCQN. Both agents start learning and playing against each other and it is noticed

that the agents start estimating rather large Q-values. The DLCQN networks start lowering their estimates, stabilizing and learning more efficiently. By contrast, DQN keeps oscillating throughout the epochs (Figure 6.4 left). Since we are interested in comparing their performance against each other during a match, we discard the first 39 epochs and consider only 40 and onwards. The reason is that as it is illustrated by the Q-values, during the first initial epochs the networks are still doing most of their learning and consequently the results from those matches are not informative. Figure 6.4 Right shows the average rewards for both agents, starting from the 40th epoch. Over those episodes, DLCQN obtains a mean average reward of 5.24 ± 7.19 and accordingly DQN obtains -5.24 ± 7.19. DLCQN best performance is registered at epoch 51. Obtaining an average reward of 19.43 and winning the 30 matches against DQN (Table 6.3). Meanwhile the best performing DQN epoch is the last one where it obtains a 6.73 average reward per match.

6.3.2 DQN vs DRUQN

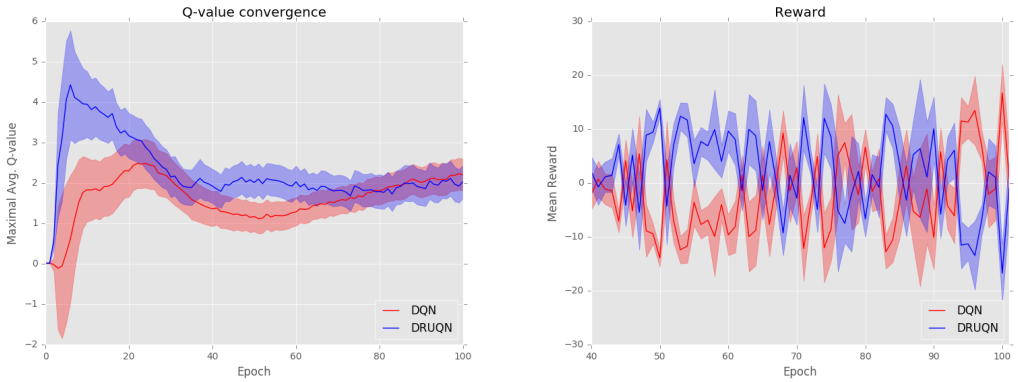

Figure 6.5 DQN vs DRUQN competitive mode. Left: Q-value convergence. Right: Reward when competing against each other.

	Mean Reward	Mean Score	Mean Score Opp.	Epoch
DQN (*vs DRUQN*)	**16.73**±5.38	19.73±1.43	**3**±4.22	101
DRUQN (*vs DQN*)	13.93±1.67	**20**	6.06±1.67	**51**

Table 6.4 DQN vs DRUQN competitive mode. First row shows the results when DQN performed the best against DRUQN. The second row is the best performing epoch of DRUQN against DQN.

The next comparison shows again a DQN agent performing worse against its multi-agent counterpart. Compared to DLCQN, DRUQN is not as dominant over DQN

(Figure 6.5 right). In fact, the largest mean reward during any epoch (after discarding the first 40), is obtained by the DQN agent (16.73). However averaging over all epochs DRUQN obtains a mean average reward of 2.18 ± 7.14 against -2.18 ± 7.14 from DQN. Similarly to DLCQN, DRUQN starts converging as it enters the 30th epoch. However DRUQN is able to maintain the Q-values at the same level until the last epoch. DQN as in the previous experiments show that the algorithm struggles to converge properly when multiple agents are involved. Figure 6.5 (left) illustrates how Q-values fluctuate throughout the learning process.

6.3.3 DRUQN vs DLCQN

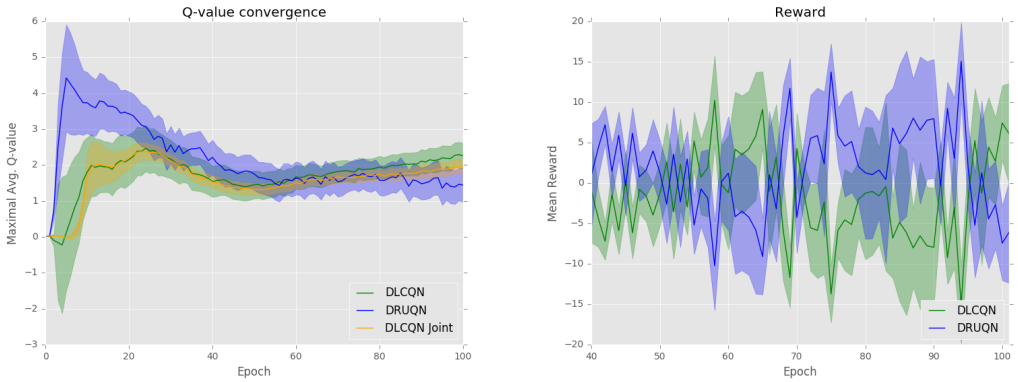

Figure 6.6 DRUQN vs DLCQN competitive mode. Left: Q-value convergence. Right: Reward when competing against each other.

	Mean Reward	Mean Score	Mean Score Opp.	Epoch
DRUQN (*vs DLCQN*)	**15.05**±4.85	**20**	**4.93**±4.85	95
DLCQN (*vs DRUQN*)	10.23±5.56	19.93±0.35	9.7±5.42	**59**

Table 6.5 DRUQN vs DLCQN competitive mode. First row shows the results when DRUQN performed the best against DLCQN. The second row is the best performing epoch of DLCQN against DRUQN.

The final comparison is between the algorithms that are better equipped to deal with non-stationarity. Figure 6.6 (left) depicts the evolution of the Q-values. After an initial period where both agents overestimate the value of the actions, they start to stabilize. Although DLCQN exhibits slight fluctuations as it was the case when it competed against DQN, it tends to stabilize faster. On the other hand, DRUQN not only stabilizes but also seems to converge. The estimates it computes remain between the same range until the last epoch. Overall DRUQN acts more efficiently, its best epoch

surpasses the performance of its DLCQN equivalent. It is also capable of winning all games and conceding less points. In addition when averaging mean rewards over all epochs, DRUQN obtains 1.70 ± 5.21 against -1.70 ± 5.21 from DLCQN.

Chapter 7

Discussion

7.1 Summary

The main focus of this dissertation was the study of decision making in the presence of other actors in changing environments. We first started by discussing Markov Decision Processes (MDP) as the theoretical foundation needed to formalize decision making under uncertainty. This also established necessary terminology such as agent, policy, reward, state or action. Policy and value iteration, the two classical approaches to solve MDPs were also briefly described. To solve MDPs one must make at least two very strong assumptions.

The first is the Markovian property which establishes that the next state depends only on the current one. MDPs presume that at any given time step an agent has full observability of the current state. In most realistic situations this presumption falls apart. In large environments, agents may be reduced to observe and gather information only within their local region. Thus the next state cannot be inferred based exclusively on the current information. Partially Observable Markov Decision Processes (POMDP) has been proposed as an extension to MDPs as a way to generalize them to describe those situations. POMDP solving is an open area of research and several subclasses have branched out to address specific niches. One of those subclasses is Decentralized Markov Decision Processes (Dec-MDP). It assumes that full observability can be attained by gathering the observations from each agent.

A second presumption of MDPs is to expect that there is already a model that can inform about the rewards and the transition probabilities. Reinforcement learning (RL) offers the possibility to circumvent this assumption. RL methods learn from trial and error, trying to discover the underlying structure of the problem by selecting actions

and observing the consequences.

One of the most popular RL algorithms in the literature is Q-Learning. It is known for its simplicity and for the convergence guarantees it provides given certain conditions. It was precisely one of the biggest concerns in this dissertation that those guarantees do not necessarily hold up in multi-agent environments. When different agents are part of the same environment, policies that were beneficial at some point may not be in the future. This type of environments are called non-stationary.

Two RL algorithms designed to deal with non-stationarity were presented. Repeated Update Q-Learning (RUQL) (Abdallah and Kaisers, 2013, 2016) and Loosely Coupled Q-Learning (LCQL) (Yu et al., 2015). These algorithms try to handle non-stationarity by focusing on different aspects. RUQL tries to fix a deficiency in Q-Learning. More specifically, in the way it estimates action values. It proposes to update the action value estimates inversely proportional to the probability of selecting an action given the policy in use. This is done in order to avoid a policy bias. It is specially relevant in non-stationary environments because we ideally want to avoid having obsolete estimations with a highly dynamic environment. Therefore in RUQL estimates for infrequent state-action pairs are updated in a larger proportion than those that are more frequent. In the case of LCQL, an explicit characterization of multiple agents is considered. The main principle is that an agent may act independently or in coordination with other agents depending on the circumstances. For this LCQL defines an independence degree, adjusted depending on the negative rewards and the observations gathered by the agent. The independence degree gives a probability for an agent to act solely on its own observations or to integrate the observations provided by other agents.

The original formulations of Q-Learning, RUQL and LCQL all assume small state spaces where individual action values are updated in a lookup table. As the state space grows this representation becomes unpractical. An overview of function approximation was provided to illustrate how RL algorithms can be used in large state spaces. Instead of learning individual action value estimates, the algorithms learn a set of parameters. Our interest focused on the recent use of deep neural networks as function approximators for RL, and more specifically in the Deep Q-Network (DQN) by Mnih et al. (2013, 2015). DQN uses the Q-Learning update rule and therefore our concerns about the shortcomings of *standard* Q-Learning in multi-agent non-stationary environments extended to DQN.

In this dissertation we introduced two new algorithms: Deep Repeated Update Q-Network (DRUQN) and Deep Loosely Coupled Q-Network (DLCQN). They are

based on RUQL and LCQL respectively, and provide a generalization to large discrete state spaces. Due to its reliance on state enumeration, for DLCQN this also involved redesigning the mechanisms to calculate an independence degree. The game Pong was used for benchmarks and three different type of tasks were devised to compare the performance among the networks. The tasks involved either competing or shared goals. In the first one, only one of the paddles is controlled by a network. This is the set up where DQN was originally tested in Mnih et al. (2013, 2015). The three networks performed roughly similar. The second task was cooperative, having as objective to keep bouncing the ball as much as possible. For this task both paddles were controlled by their own network and learning occured simultaneously. DRUQN was the best performer however DLCQN was the worst. In the final task the algorithms were tested against each other in a competitive scenario. Overall DRUQN and DLCQN surpassed DQN in their general performance on this task. Nonetheless DQN obtained a better individual epoch than any of those reported by DRUQN.

7.2 Observations

7.2.1 Generality

A driving force motivating the development of DQN is the search for increasingly general algorithms. Using the same architecture DQN learned to play various games with distinct themes and goals. An important aspect of the algorithms introduced in this dissertation is that they extend further the domain of application. They can be applied to single-agent as well as multi-agent environments. For DRUQL, a modification in the effective learning rate is what grants it the capacity to deal with non-stationary environments regardless of their source. In the case of DLCQN, the independence degree decides probabilistically whether to act in concert with the information coming from another agent or individually. Although unlike DRUQL, DLCQN incorporates observations from another agent for decision making. There is by no means a requirement that another agent must exist in the same environment. Thus, in single agent environments the independence degree will remain at $\xi = 1$. It is only when another agent is detected, that information sharing may occur. Then the agent decides whether or not is convenient to use that information for decision making. Only then, the necessary adjustment to its independence degree is made.

7.2.2 Non-Stationarity

Comparisons to DQN, favored DRUQN in the second and third task and DLCQN in the third. These tasks involved two agents learning concurrently. They were conceived with the intention of testing the algorithms in their ability to deal with a changing environment due to the presence of other actors. Nonetheless the results against DQN are far from overwhelming. There are possible reasons that may explain these findings.

Divergence and instability was commonly observed in early use of function approximation with reinforcement learning (Geramifard, 2013; Tsitsiklis and B. V. Roy, 1997). Consequently, much emphasis and attention has been invested in the creation of stable architectures that can integrate reinforcement learning techniques (Hausknecht et al., 2014; Koutnik et al., 2013; Riedmiller, 2005). DQN for instance, incorporated *frame skipping*, *reward clipping*, *replay memory* and the option to update the target weights only after a given number of iterations. In particular, replay memory and freezing the target weights help to break correlations between data because they force to update the estimations with respect to non-sequential observations. Replay memory for instance involves selecting from samples that occurred when past policies could have been more efficient but that no longer apply. This is in principle something that DRUQN tries to address by modifying the update rule. Therefore although in DQN the update rule remains the same, the architecture itself has mechanisms to counteract to an extent policy oscillations and data distribution biases.

A second factor, albeit one that only affects DRUQN, is the choice of the optimizer. In DRUQL (or RUQL) for every update, an effective learning rate is computed with respect to the selected action and the current policy. For lookup tables or stochastic gradient descent this is trivially implemented because only one update is done at a time. If using standard batch or minibatch gradient descent, a global learning rate determines the magnitude of the update. In order to implement it using modern libraries this requires setting the global learning rate to 1 and instead taking each gradient individually and rescale it according to its custom effective learning rate. More problematic is the use of gradient descent optimizers that determine adaptive learning rates based on moving averages such as Adam, Adadelta, Adagrad or RMSProp. As described in the methods section, Adam was used for all experiments because it provides state of the art gradient based learning (Kingma and Ba, 2014). Due to the computationally and time intensive nature of our experimental tasks, it was decided to use optimized libraries that could provide such methods. This also meant that granular access was not easily available.

This was a limitation that was overcome by developing a practical implementation that could provide an individual approximation of each gradients before they were passed to the optimizer (Appendix B). Nonetheless, these approximations could have lead to suboptimal performance by our DRUQL implementation.

Another issue, this one observed with DLCQN was the large difference in performance it had between the cooperative task and the competitive ones. It could be speculated that since in cooperative mode, rewards are shared between the agents, they are forced to revise its degree of independence regardless of whether or not they complied with their part of the task. This could have provoked that the agents attempt to cooperate by actively being in each other field of vision which might not be necessarily beneficial for a game such as Pong due to the trajectories the ball takes. Let us remember that we consider that an agent is observed by another when at least part of the paddle is horizontally in parallel with the other. This criteria was defined by taking inspiration on the original LCQL algorithm where a robot navigation task was solved. However whereas in navigation tasks it could be more intuitive to define when an agent is observed, in other tasks such as Pong, it may not be as unambiguous. Especially when considering that in the competitive tasks this same criteria did not affect its performance against the other algorithms.

Last, Pong was one of the games where DQN routinely found successful policies and attained full scores in Mnih et al. (2013, 2015). Thus, perhaps the tasks were just not complex enough. Adding another network controlled agent did not seem to affect its performance greatly. As it has been explained above this could be due to the stability mechanisms built into the architecture. Therefore future evaluations and comparisons in multi-agent reinforcement learning should consider more complex environments and tasks.

7.2.3 DRUQN or DLCQN

Between the two algorithms presented in this dissertation, there are advantages and disadvantages that can be distinguished. DRUQN for example, handles non-stationarity via effective learning rates. Therefore there is no need to define specific hand-crafted independence criteria that could affect the performance depending of the modality of the task. As it was observed in the experiments, DRUQN performed well for the three set of tasks. On the other hand for DLCQN, the criteria used to adjust the independence degree is different from task to task. DLCQN also introduces other parameters such as λ and

the time steps required to update ψ, which implies additional fine tuning. Furthermore, the original algorithm, LCQL, makes strong assumptions on the ability to list the states. Which in practice is only feasible for small state spaces. For DLCQN we have limited this reliance on state listing by defining a single independence degree ξ, ψ and ε for all states. However a better approach could have been to use an additional function approximator to obtain the independence degree instead of relying on intermediate variables ψ, ε and max_{temp}. DLCQN is more computationally and time intensive than both DQN and DRUQN as it requires $n+1$ neural networks where n is the number of agents. The extra network corresponds to the channel that approximates the global Q-function when incoming information from other agents is available. Another potential issue is that the joint state grows as the number of agents increases. This translates into an additional challenge of designing more compact state representations.

As we have noted previously, DRUQN relies on computing individual effective learning rates which might not be easily implemented with adaptive optimizers. Another disadvantage of DRUQN is that it tries to address a problem in the way Q-Learning estimates action values. However this may not necessarily extend to other reinforcement learning algorithms. Meanwhile, the main idea behind DLCQN is to learn where to coordinate. This approach can be easily adapted to other reinforcement learning algorithms.

7.3 Future Work

Many improvements to DQN have been proposed since its introduction (Section 2.3.1.2). None of these extensions were incorporated into the project as we wanted to keep the scope narrowed to the analysis of the algorithms in multi-agent non-stationary environments. Extensions such prioritized experience replay (Schaul et al., 2015) or dueling architecture (Wang et al., 2015) are equally applicable to DRUQN or DLCQN. Future work could consider increasing the performance of the algorithms with those modifications.

In the third task, it was observed that the Q-values estimated by DLCQN and DQN fluctuated. In comparison, DRUQN was more stable against both. Future work could combine both approaches and integrate DLCQN with DRUQN's update rule, providing an additional layer of stability. Alternatively DLCQN could also use any of the other suggested modifications to the Q-Learning estimator such as bias corrected Q-learning (Lee and Powell, 2012) or Double Q-Learning (van Hasselt, 2010).

We have discussed that the choice in the criteria for adjusting the independence degree in DLCQN will impact its performance. Thus its design should be subject of careful consideration for future research. In addition, the current form of DLCQN only adjusts the independence degree when a *conflict* or a negative reward is received. Coordinated behavior, however could also benefit from analyzing when a positive reward is also received and ponder whether or not this could be a consequence of cooperating adequately. Future work could follow in this direction. One of the challenges of extending LCQL to large state spaces was its extreme reliance on state enumeration. For each state of every agent there is an independence degree. In DLCQN this was reduced to a single independence degree per agent. This independence degree is then used and adjusted for all states. However a much better solution would have been to determine the independence degree through a function approximator providing a much better generalization.

Finally, it was suggested that if the tasks and the environment in which the algorithms were tested had been more complex it could have lead to more significant results. Potentially more conclusive analyses could come from studying the algorithms in games providing a much richer range of interactions and situations such *Warlords*, *Combat* or *Armor Ambush*.

Bibliography

Abdallah, Sherief and Michael Kaisers (2013). "Addressing the Policy-bias of Q-learning by Repeating Updates". In: *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*. AAMAS '13. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, pp. 1045–1052.

Abdallah, Sherief and Michael Kaisers (2016). "Addressing Environment Non-Stationarity by Repeating Q-learning Updates". In: *Journal of Machine Learning Research* 17.46, pp. 1–31.

Abel, David, Alekh Agarwal, Fernando Diaz, Akshay Krishnamurthy, and Robert E. Schapire (2016). "Exploratory Gradient Boosting for Reinforcement Learning in Complex Domains". In: *arXiv:1603.04119 [cs, stat]*.

Agogino, Adrian K. and Kagan Tumer (2005). "Quicker Q-Learning in Multi-Agent Systems".

Allen, Martin and Shlomo Zilberstein (2009). "Complexity of Decentralized Control: Special Cases". In: pp. 19–27.

Asmuth, John, Lihong Li, Michael L. Littman, Ali Nouri, and David Wingate (2012). "A Bayesian Sampling Approach to Exploration in Reinforcement Learning". In: *arXiv:1205.2664 [cs]*.

Åström, K. J (1965). "Optimal control of Markov processes with incomplete state information". In: *Journal of Mathematical Analysis and Applications* 10.1, pp. 174–205.

Barto, Andrew G., R. S. Sutton, and C. W. Anderson (1983). "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.5, pp. 834–846.

Bellemare, Marc G., Yavar Naddaf, Joel Veness, and Michael Bowling (2012). "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *arXiv:1207.4708 [cs]*.

Bellman, Richard (1954). "The theory of dynamic programming". EN. In: *Bulletin of the American Mathematical Society* 60.6, pp. 503–515.

Bellman, Richard (1956). "Dynamic Programming and Lagrange Multipliers". In: *Proceedings of the National Academy of Sciences of the United States of America* 42.10, pp. 767–769.

Bengio, Yoshua, Pascal Lamblin, Dan Popovici, Hugo Larochelle, et al. (2007). "Greedy layer-wise training of deep networks". In: *Advances in neural information processing systems* 19, p. 153.

Bernstein, Daniel S., Shlomo Zilberstein, and Neil Immerman (2000). "The Complexity of Decentralized Control of Markov Decision Processes". In: *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*. UAI'00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 32–37.

Bordes, Antoine, Sumit Chopra, and Jason Weston (2014). "Question Answering with Subgraph Embeddings". In: *arXiv:1406.3676 [cs]*.

Bowling, Michael (2000). “Convergence Problems of General-Sum Multiagent Reinforcement Learning”. In: *IN PROCEEDINGS OF THE SEVENTEENTH INTERNATIONAL CONFERENCE ON MACHINE LEARNING*. Morgan Kaufmann, pp. 89–94.

Buşoniu, Lucian, Robert Babuška, and Bart De Schutter (2008). “A comprehensive survey of multiagent reinforcement learning”. In: *IEEE Transactions on Systems, Man, And Cybernetics-Part C: Applications and Reviews, 38 (2), 2008*.

Buşoniu, Lucian, Robert Babuška, and Bart De Schutter (2010). “Multi-agent Reinforcement Learning: An Overview”. en. In: *Innovations in Multi-Agent Systems and Applications - 1*. Ed. by Dipti Srinivasan and Lakhmi C. Jain. Studies in Computational Intelligence 310. Springer Berlin Heidelberg, pp. 183–221.

Claus, Caroline and Craig Boutilier (1998). “The Dynamics of Reinforcement Learning in Cooperative Multiagent Systems”. In: *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*. AAAI ’98/IAAI ’98. Menlo Park, CA, USA: American Association for Artificial Intelligence, pp. 746–752.

Collobert, Ronan, Jason Weston, Leon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa (2011). “Natural Language Processing (almost) from Scratch”. In: *arXiv:1103.0398 [cs]*.

Crites, Robert H. and Andrew G. Barto (1998). “Elevator Group Control Using Multiple Reinforcement Learning Agents”. In: *Mach. Learn.* 33.2-3, pp. 235–262.

Dayan, Peter and Yael Niv (2008). “Reinforcement learning: the good, the bad and the ugly”. eng. In: *Current Opinion in Neurobiology* 18.2, pp. 185–196.

De Hauwere, Yann-Michaël, Peter Vrancx, and Ann Nowé (2009). *Learning what to observe in Multi-agent Systems*.

De Hauwere, Yann-Michaël, Peter Vrancx, and Ann Nowé (2010). “Learning Multi-agent State Space Representations”. In: *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*. AAMAS ’10. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, pp. 715–722.

Egorov, Maxim (2015). “Deep Reinforcement Learning with POMDPs”. In:

Egorov, Maxim (2016). “Multi-Agent Deep Reinforcement Learning”. In:

Foerster, Jakob N., Yannis M. Assael, Nando de Freitas, and Shimon Whiteson (2016a). “Learning to Communicate to Solve Riddles with Deep Distributed Recurrent Q-Networks”. In: *arXiv:1602.02672 [cs]*.

Foerster, Jakob N., Yannis M. Assael, Nando de Freitas, and Shimon Whiteson (2016b). “Learning to Communicate with Deep Multi-Agent Reinforcement Learning”. In: *arXiv:1605.06676 [cs]*.

Fukushima, Kunihiko (1980). “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. en. In: *Biological Cybernetics* 36.4, pp. 193–202.

Geramifard, Alborz (2013). “A Tutorial on Linear Function Approximators for Dynamic Programming and Reinforcement Learning”. en. In: *Foundations and Trends® in Machine Learning* 6.4, pp. 375–451.

Ghavamzadeh, Mohammad, Sridhar Mahadevan, and Rajbala Makar (2006). “Hierarchical multi-agent reinforcement learning”. en. In: *Autonomous Agents and Multi-Agent Systems* 13.2, pp. 197–229.

Goodfellow, Ian J., Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio (2014). "Generative Adversarial Networks". In: *arXiv:1406.2661 [cs, stat]*.

Guestrin, Carlos, Michail G. Lagoudakis, and Ronald Parr (2002). "Coordinated Reinforcement Learning". In: *Proceedings of the Nineteenth International Conference on Machine Learning*. ICML '02. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 227–234.

Hausknecht, Matthew, J. Lehman, R. Miikkulainen, and P. Stone (2014). "A Neuroevolution Approach to General Atari Game Playing". In: *IEEE Transactions on Computational Intelligence and AI in Games* 6.4, pp. 355–366.

Hausknecht, Matthew and Peter Stone (2015). "Deep Recurrent Q-Learning for Partially Observable MDPs". In: *arXiv:1507.06527 [cs]*.

Hinton, Geoffrey et al. (2012). "Deep Neural Networks for Acoustic Modeling in Speech Recognition". In:

Hochreiter, S. and Juergen Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780.

Howard, Ronald A (1960). *Dynamic programming and Markov processes*. English. OCLC: 523881. Cambridge: Technology Press of Massachusetts Institute of Technology.

Hu, Junling and Michael P. Wellman (1998). "Multiagent Reinforcement Learning: Theoretical Framework and an Algorithm". In: *Proceedings of the Fifteenth International Conference on Machine Learning*. ICML '98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 242–250.

Jean, Sébastien, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio (2014). "On Using Very Large Target Vocabulary for Neural Machine Translation". In: *arXiv:1412.2007 [cs]*.

Jung, Tobias and Peter Stone (2012). "Gaussian Processes for Sample Efficient Reinforcement Learning with RMAX-like Exploration". In: *arXiv:1201.6604 [cs]*.

Kaelbling, Leslie Pack, Michael L. Littman, and Andrew W. Moore (1996). "Reinforcement Learning: A Survey". In: *J. Artif. Int. Res.* 4.1, pp. 237–285.

Kaisers, Michael and Karl Tuyls (2010). "Frequency Adjusted Multi-agent Q-learning". In: *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*. AAMAS '10. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, pp. 309–316.

Kapetanakis, Spiros and Daniel Kudenko (2005). "Reinforcement Learning of Coordination in Heterogeneous Cooperative Multi-agent Systems". In: *Adaptive Agents and Multi-Agent Systems II*. Ed. by David Hutchison et al. Vol. 3394. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 119–131.

Kingma, Diederik and Jimmy Ba (2014). "Adam: A Method for Stochastic Optimization". In: *arXiv:1412.6980 [cs]*.

Kok, Jelle R., Eter Jan Hoen, Bram Bakker, and Nikos Vlassis (2005). "Utile coordination: Learning interdependencies among cooperative agents". en. In:

Kok, Jelle R. and Nikos Vlassis (2004). "Sparse Cooperative Q-learning". In: *Proceedings of the Twenty-first International Conference on Machine Learning*. ICML '04. New York, NY, USA: ACM, pp. 61–.

Konidaris, George and Sarah Osentoski (2008). *Value function approximation in reinforcement learning using the Fourier basis*. Tech. rep.

Koutnik, Jan, Giuseppe Cuccu, Juergen Schmidhuber, and Faustino Gomez (2013). "Evolving Large-scale Neural Networks for Vision-based Reinforcement Learning". In: *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*. GECCO '13. New York, NY, USA: ACM, pp. 1061–1068.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*, pp. 1097–1105.

Lagoudakis, Michail G. and Ronald Parr (2003). "Least-Squares Policy Iteration". In: *Journal of Machine Learning Research* 4.Dec, pp. 1107–1149.

Lauer, Martin and Martin Riedmiller (2000). "An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems". In: *ResearchGate*.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". en. In: *Nature* 521.7553, pp. 436–444.

Lee, Donghun and Warren B. Powell (2012). "An Intelligent Battery Controller Using Bias-corrected Q-learning". In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*. AAAI'12. Toronto, Ontario, Canada: AAAI Press, pp. 316–322.

Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra (2015). "Continuous control with deep reinforcement learning". In: *CoRR* abs/1509.02971.

Lin, Long-ji (1992). "Self-improving reactive agents based on reinforcement learning, planning and teaching". In: *Machine Learning*, pp. 293–321.

Littman, Michael L. (1994). "Markov games as a framework for multi-agent reinforcement learning". In: *IN PROCEEDINGS OF THE ELEVENTH INTERNATIONAL CONFERENCE ON MACHINE LEARNING*. Morgan Kaufmann, pp. 157–163.

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

Littman, Michael L. (2001b). "Value-function reinforcement learning in Markov games". en. In: *Cognitive Systems Research* 2.1, pp. 55–66.

Littman, Michael L., Anthony R. Cassandra, and Leslie Pack Kaelbling (1995). *Learning policies for partially observable environments: Scaling up*.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller (2013). "Playing Atari with Deep Reinforcement Learning". In: *arXiv:1312.5602 [cs]*.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". en. In: *Nature* 518.7540, pp. 529–533.

Mohamed, A., G. E. Dahl, and G. Hinton (2012). "Acoustic Modeling Using Deep Belief Networks". In: *Trans. Audio, Speech and Lang. Proc.* 20.1, pp. 14–22.

Nair, Vinod and Geoffrey E. Hinton (2010). "Rectified Linear Units Improve Restricted Boltzmann Machines". In: pp. 807–814.

Nash, John F. (1950). "Equilibrium points in n-person games". en. In: *Proceedings of the National Academy of Sciences* 36.1, pp. 48–49.

Osband, Ian, Benjamin Van Roy, and Zheng Wen (2014). “Generalization and Exploration via Randomized Value Functions”. In: *arXiv:1402.0635 [cs, stat]*.

Panait, Liviu and Sean Luke (2005). “Cooperative Multi-Agent Learning: The State of the Art”. In: *Autonomous Agents and Multi-Agent Systems* 11.3, pp. 387–434.

Puterman, Martin L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. 1st. New York, NY, USA: John Wiley & Sons, Inc.

Pyeatt, Larry D. and Adele E. Howe (1998). *Decision Tree Function Approximation in Reinforcement Learning*. Tech. rep. In Proceedings of the Third International Symposium on Adaptive Systems: Evolutionary Computation and Probabilistic Graphical Models.

Pynadath, David V. and Milind Tambe (2002). “Multiagent Teamwork: Analyzing the Optimality and Complexity of Key Theories and Models”. In: *In First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS’02*, pp. 873–880.

Riedmiller, Martin (2000). *Concepts and Facilities of a Neural Reinforcement Learning Control Architecture for Technical Process Control*.

Riedmiller, Martin (2005). “Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method”. en. In: *Machine Learning: ECML 2005*. Ed. by Joao Gama, Rui Camacho, Pavel B. Brazdil, Alipio Mario Jorge, and Luis Torgo. Lecture Notes in Computer Science 3720. Springer Berlin Heidelberg, pp. 317–328.

Roth, Maayan, Reid Simmons, and Manuela Veloso (2007). “Exploiting Factored Representations for Decentralized Execution in Multiagent Teams”. In: *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS ’07. New York, NY, USA: ACM, 72:1–72:7.

Roy, Nicholas (2003). “Finding Approximate POMDP Solutions Through Belief Compression”. AAI3102464. PhD thesis. Pittsburgh, PA, USA: Carnegie Mellon University.

Roy, Nicholas and Sebastian Thrun (1999). “Coastal Navigation with Mobile Robots”. In: *In Advances in Neural Processing Systems 12*, pp. 1043–1049.

Rummery, G. A. and M. Niranjan (1994). *On-Line Q-Learning Using Connectionist Systems*. Tech. rep.

Schaul, Tom, John Quan, Ioannis Antonoglou, and David Silver (2015). “Prioritized Experience Replay”. In: *arXiv:1511.05952 [cs]*.

Schmidhuber, Juergen (2015). “Deep Learning in Neural Networks: An Overview”. In: *Neural Networks* 61, pp. 85–117.

Schultz, W., P. Dayan, and P. R. Montague (1997). “A neural substrate of prediction and reward”. eng. In: *Science (New York, N.Y.)* 275.5306, pp. 1593–1599.

Silver, David, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller (2014). “Deterministic Policy Gradient Algorithms”. In: pp. 387–395.

Sorokin, Ivan, Alexey Seleznev, Mikhail Pavlov, Aleksandr Fedorov, and Anastasiia Ignateva (2015). “Deep Attention Recurrent Q-Network”. In: *arXiv:1512.01693 [cs]*.

Sukhbaatar, Sainbayar, Arthur Szlam, and Rob Fergus (2016). “Learning Multiagent Communication with Backpropagation”. In: *arXiv:1605.07736 [cs]*.

Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le (2014). “Sequence to Sequence Learning with Neural Networks”. In: *arXiv:1409.3215 [cs]*.

Sutton, Richard S. (1988). “Learning to predict by the methods of temporal differences”. In: *MACHINE LEARNING*. Kluwer Academic Publishers, pp. 9–44.

Sutton, Richard S. and Andrew G. Barto (1998). *Reinforcement learning: An introduction*. Vol. 28. MIT press.

Tampuu, Ardi, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente (2015). “Multiagent Cooperation and Competition with Deep Reinforcement Learning”. In: *arXiv:1511.08779 [cs, q-bio]*.

Tan, Ming (1993). “Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents”. In: *In Proceedings of the Tenth International Conference on Machine Learning*. Morgan Kaufmann, pp. 330–337.

Tesauro, Gerald (1995). “Temporal Difference Learning and TD-Gammon”. In: *Commun. ACM* 38.3, pp. 58–68.

Tesauro, Gerald (2003). “Extending Q-Learning to General Adaptive Multi-Agent Systems”. In: *IN ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 16*. MIT Press, p. 2004.

Thrun, Sebastian (2000). “Monte Carlo POMDPs”. In: *Advances in Neural Information Processing Systems 12*. Ed. by S. A. Solla, T. K. Leen, and K. Müller. MIT Press, pp. 1064–1070.

Thrun, Sebastian and Anton Schwartz (1993). “Issues in Using Function Approximation for Reinforcement Learning”. In: *IN PROCEEDINGS OF THE FOURTH CONNECTIONIST MODELS SUMMER SCHOOL*.

Tsitsiklis, J. N. and B. Van Roy (1997). “An analysis of temporal-difference learning with function approximation”. In: *IEEE Transactions on Automatic Control* 42.5, pp. 674–690.

Tuyls, Karl and Gerhard Weiss (2012). “Multiagent Learning: Basics, Challenges, and Prospects”. en. In: *AI Magazine* 33.3, p. 41.

van Hasselt, Hado (2010). “Double Q-learning”. In: *Advances in Neural Information Processing Systems 23*. Ed. by J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta. Curran Associates, Inc., pp. 2613–2621.

van Hasselt, Hado, Arthur Guez, and David Silver (2015). “Deep Reinforcement Learning with Double Q-learning”. In: *arXiv:1509.06461 [cs]*.

Von Neumann, J. and O. Morgenstern (1944). *Theory of games and economic behavior*. Vol. xviii. Princeton, NJ, US: Princeton University Press.

Wang, Ziyu, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas (2015). “Dueling Network Architectures for Deep Reinforcement Learning”. In: *arXiv:1511.06581 [cs]*.

Watkins, Christopher J. C. H. (1989). “Learning from delayed rewards”. PhD thesis. King’s College, Cambridge.

Watkins, Christopher J. C. H. and Peter Dayan (1992). “Q-learning”. In: *Machine Learning*, pp. 279–292.

Williams, Ronald J. (1992). “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning”. In: *Mach. Learn.* 8.3-4, pp. 229–256.

Yu, Chao, Minjie Zhang, Fenghui Ren, and Guozhen Tan (2015). “Multiagent Learning of Coordination in Loosely Coupled Multiagent Systems”. eng. In: *IEEE transactions on cybernetics* 45.12, pp. 2853–2867.

Appendix A

DRUQN Update Rule

In order to obtain Equation 4.1 from,

$$Q^{t+1}(s,a) = [1-\alpha]^{\frac{1}{\pi(s,a)}} Q(s,a) + [1-(1-\alpha)^{\frac{1}{\pi(s,a)}}][r + \gamma \max_a Q(s',a')] \tag{A.1}$$

Substitute $z_{\pi(s,a)} = 1-(1-\alpha)^{\frac{1}{\pi(s,a)}}$,

$$Q^{t+1}(s,a) = [1-\alpha]^{\frac{1}{\pi(s,a)}} Q(s,a) + z_{\pi(s,a)}[r + \gamma \max_a Q(s',a')] \tag{A.2}$$

Then from $z_{\pi(s,a)}$ let $\omega = (1-\alpha)^{\frac{1}{\pi(s,a)}}$. Therefore $z_{\pi(s,a)} = 1-\omega$, substituting in Equation A.2,

$$Q^{t+1}(s,a) = \omega Q(s,a) + [(1-\omega)(r + \gamma \max_a Q(s',a'))] \tag{A.3}$$

Since $\omega = 1 - z_{\pi(s,a)}$ then,

$$Q^{t+1}(s,a) = (1-z_{\pi(s,a)})Q(s,a) + [(1-(1-z_{\pi(s,a)}))(r + \gamma \max_a Q(s',a'))] \tag{A.4}$$

Simplifying further,

$$Q^{t+1}(s,a) = Q(s,a) - z_{\pi(s,a)}Q(s,a) + z_{\pi(s,a)}[r + \gamma \max_a Q(s',a')] \tag{A.5}$$

Finally, rearrange as,

$$Q^{t+1}(s,a) = Q(s,a) + z_{\pi(s,a)}[r + \gamma \max_a Q(s',a') - Q(s,a)] \tag{A.6}$$

Appendix B

DRUQN Approximation with ADAM

Gradient descent considers a parameter update rule of the form,

$$\theta_{i+1} \leftarrow \theta_i + \alpha \nabla_{\theta_i} L(\theta_i) \tag{B.1}$$

For Adam (Kingma and Ba, 2014) the update rule is given by,

$$\theta_{t+1} \leftarrow \theta_t + \frac{\alpha}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t \tag{B.2}$$

Where,

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{B.3}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{B.4}$$

Are biased corrected estimates of the first and second moment, β_1, β_2 are exponentially decaying rates and,

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{B.5}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{B.6}$$

Represent first and second moment that stores exponentially decaying averages of past gradients g_t. For DRUQN these averages are obtained by,

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_{\theta_i} L_i(\theta_i) \tag{B.7}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \nabla_{\theta_i}^2 L_i(\theta_i) \tag{B.8}$$

Where $\nabla_{\theta_i} L_i(\theta_i)$ is the gradient of the loss function,

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[(y_i - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)] \tag{B.9}$$

For DRUQN instead of a global α rescaling all gradients, for each gradient l there is an effective learning rate $z^{(l)}_{\pi(s,a)}$. An issue then is faced if using an optimized tensor library accepting only a single α. In standard gradient descent this can be overcome by setting $\alpha = 1$ and multiplying the effective learning rate with the elements of $\nabla_{\theta_i} L_i(\theta_i)$,

$$\alpha z^{(l)}_{\pi(s,a)} \nabla_{\theta_i^{(l)}} L_i(\theta_i^{(l)}) = \mathbb{E}_{(s,a,r,s',z_{\pi(s,a)})\sim U(D)}[(z^{(l)}_{\pi(s,a)} y_i - z^{(l)}_{\pi(s,a)} Q(s,a;\theta_i^{(l)}))\nabla_{\theta_i^{(l)}} Q(s,a;\theta_i^{(l)})] \tag{B.10}$$

In the case of Adam, the existence of other parameters does not permit this possibility. Modifying the library to accept an array of $\alpha_{(l)}$ is left as a future extension. However, an approximation was obtained by setting the effective learning rate to $\eta^{(l)}_{\pi(s,a)} = \frac{z^{(l)}_{\pi(s,a)}}{\alpha}$. From Equation B.5 and B.6 this modification leads to,

$$m_t^{(l)} = \beta_1^{(l)} m_{t-1} + (1-\beta_1)\eta^{(l)}_{\pi(s,a)} g_t \tag{B.11}$$

$$v_t^{(l)} = \beta_2^{(l)} v_{t-1} + (1-\beta_2)\eta^{(l)}_{\pi(s,a)} g_t^2 \tag{B.12}$$

Substituting in B.2,

$$\theta^{(l)}_{t+1} \leftarrow \theta^{(l)}_t + \frac{\alpha \dfrac{m_t^{(l)}}{1-\beta_1^t}}{\sqrt{\beta_2^{(l)} v_{t-1} + (1-\beta_2)\eta^{(l)}_{\pi(s,a)} g_t^2} + \varepsilon} \tag{B.13}$$

$$\theta^{(l)}_{t+1} \leftarrow \theta^{(l)}_t + \frac{\dfrac{\alpha\beta_1^{(l)} m_{t-1} + z^{(l)}_{\pi(s,a)}(1-\beta_1) g_t}{1-\beta_1^t}}{\sqrt{\beta_2^{(l)} v_{t-1} + (1-\beta_2)\eta^{(l)}_{\pi(s,a)} g_t^2} + \varepsilon} \tag{B.14}$$

Which is then applied to update the network parameters.

Appendix C

Full List of Network Parameters

Minibatch size	200
Replay memory size D	500,000
Frames given as input	4
Target network update frequency	2
Discount factor γ	0.99
Frame skipping k	4
Learning rate α	0.000001
Adam β_1	0.9
Adam β_2	0.999
Adam ε	1×10^{-8}
Initial exploration ϵ	1
Final exploration ϵ	0.05
Exploration frames	500,000
Replay start size	50,000
Initial independence degree ξ	1
Initial coordinated learning ψ	0
Initial coordinated learning ε	0
Coordinated learning decay rate λ	0.8
Independence degree update frequency	250