Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# Accelerating Learning in Multiagent Domains through Experience Sharing

Lucas Oliveira Souza

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientadora
Prof.ª Dr.ª Célia Ghedini Ralha

Brasília
2019

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenador: Prof. Dr. Bruno Luiggi Macchiavello Espinoza

Banca examinadora composta por:

Prof.ª Dr.ª Célia Ghedini Ralha (Orientadora) — CIC/UnB
Prof. Dr. Teófilo Emidio de Campos — CIC/UnB
Prof.ª Dr.ª Anna Helena Reali Costa — USP

**CIP — Catalogação Internacional na Publicação**

Souza, Lucas Oliveira.

Accelerating Learning in Multiagent Domains through Experience Sharing / Lucas Oliveira Souza. Brasília : UnB, 2019.
87 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2019.

1. aprendizado de reforço, 2. aprendizado de reforço profundo,
3. aprendizado de reforço multiagente, 4. transferência de aprendizado,
5. compartilhamento de conhecimento

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# Accelerating Learning in Multiagent Domains through Experience Sharing

Lucas Oliveira Souza

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Prof.ª Dr.ª Célia Ghedini Ralha (Orientadora)
CIC/UnB

Prof. Dr. Teófilo Emidio de Campos
CIC/UnB

Prof.ª Dr.ª Anna Helena Reali Costa
USP

Prof. Dr. Bruno Luiggi Macchiavello Espinoza
Coordenador do Mestrado em Informática

Brasília, 15 de Fevereiro de 2019

# Acknowledgments

It is not an easy journey to go from practitioner to researcher, specially later in life. These last two years have been as challenging as rewarding, and I have many to thank for in this journey. I feel this is just the first step into a much longer path towards being an active and relevant contributor to the scientific community.

My family, as always, is the first line of support. My parents, siblings, lovely nephews, and my wife Eline. My close friends are my extended family, and have been there for me as long as I can remember.

The meetings with my colleagues from UnB were always fun but productive. We've learned a lot together, attending conferences, going through competitions, presentations and tight deadlines, always stretching out to see how far we could reach.

I'm much obliged to my employer, the Brazilian Office of the Comptroller General. Fighting corruption is a noble goal, and I'm proud of the work I've done there. I've been on non-paid leave during most of the last two years, and my supervisors André and Matheus have been very supportive during this period.

In the last years I've also part-time mentored students at online machine learning and data science courses at Udacity. I'm really thankful for all the students I had the opportunity to mentor, it has been an amazing learning experience for me. During the countless evenings we spent together, they pushed me to learn more, research more, and rethink what I thought I knew. Some of these students became close friends I will take for life, and it makes me truly happy to see they land their dream jobs and fulfill their aspirations.

Prof. Ann Nowé and the organizers of the 2017 ACAI Summer School on Reinforcement Learning done a superb job in organizing this learning experience and were kind enough to let me attend. I shared nine intense days with very smart colleagues, who taught and inspired me. It sparkled my interest into RL to the point I've decided further researching on it is the next step I wish to take in my career. During a short visit to VUB many researcher colleagues gave me support to streamline my proposal, specially Prof. Gabriel Ramos, who has been a good friend and mentor.

Prof. Anna Reali and Prof. Teófilo de Campos reviewed my initial proposal and steered me towards the right direction. When drafting my research project I had great ambitions, and they wisely made me realize early on there was so much I could do with the resources and time I had available. Prof. Li Weigang taught me a lot about the scientific method and inner workings of the research community.

A special thanks to my advisor, Prof. Célia Ghedini Ralha, who has encouraged and supported me from the beginning, with enormous patience, wisdom and knowledge. She was the one who first convinced me that going through a Masters program would be a good experience, and as always, she was right. This work is as much mine as it is hers.

# Resumo Expandido

Essa dissertação contribui para o crescente campo de inteligência artificial e aprendizado de máquina. Aprendizado é um componente essencial do comportamento humano, a faculdade por trás da nossa habilidade de se adaptar. E essa característica única que diferencia seres humanos de outras espécies, e nos permitiu perserverar e dominar o mundo como nos conhecemos. Através de algoritmos de aprendizado, nós buscamos imbuir agentes artificiais com essa mesma capacidade, para que eles possam aprender e se adaptar interagindo com o ambiente, conseguindo desta forma aumentar seu potencial de atingir seus objetivos.

Nesse trabalho, nós buscamos resolver o problema de como múltiplos agentes cooperativos aprendendo concomitantemente podem se beneficar de conhecimento compartilhado entre eles. A habilidade de compartilhar conhecimento adquirido, seja instantaneamente ou através de gerações, é peça chave para a nossa evolução. Segue que o compartilhamento de conhecimento entre agentes autônomos pode ser a chave para acelerar conhecimento em sistemas multiagentes cooperativos. Baseado nesse raciocínio, neste trabalho investigamos métodos de compartilhamento de conhecimento que pode efetivamente levar a uma aceleração no aprendizado.

A pesquisa é focada na abordagem de transferência de conhecimento através do compartilhamento de experiências. O modelo MultiAgent Cooperative Experience Sharing (MACES) define uma arquitetura que permite troca de experiências entre agentes cooperativos aprendendo concomitantemente. Neste modelo, investigamos diferentes métodos de compartilhamento de experiências que podem levar a aceleração do aprendizado.

O modelo é validado em dois problemas diferentes de aprendizado de reforço, um problema de controle clássico e um de navegação. Os resultados apresentados mostram que o MACES é capaz de reduzir em mais da metade o número de episódios necessários para completar uma tarefa através da cooperação de apenas dois agentes, comparado a agentes não cooperativos. O modelo é aplicável a agentes que implementam métodos de aprendizado de reforço profundo.

**Palavras-chave:** aprendizado de reforço, aprendizado de reforço profundo, aprendizado de reforço multiagente, transferência de aprendizado, compartilhamento de conhecimento

# Extended Abstract

This dissertation is a contribution to the burgeoning field of artificial intelligence and machine learning. Learning is a core component of human behaviour, the faculty behind our ability to adapt. It is the single characteristic that differentiate humans from other species, and has allowed us to persevere and dominate the world as we know. Through learning algorithms, we seek to imbue artificial agents with the same capacity, so they can as well learn and adapt by interacting with the environment, thus enhancing their potential to achieve their goals.

In this work, we address the hard problem of how multiple cooperative agents learning concurrently to achieve a goal can benefit from sharing knowledge with each other. Key to our evolution is our ability to share learned knowledge with each other instantaneously and through generations. It follows that knowledge sharing between autonomous and independent agents could as well become the key to accelerate learning in cooperative multiagent settings. Pursuing this line of inquiry, we investigate methods of knowledge sharing that can effectively lead to faster learning.

We focus on the approach of transferring knowledge by experience sharing. The proposed MultiAgent Cooperative Experience Sharing (MACES) model defines an architecture that allows experience sharing between concurrently learning cooperative agents. Within MACES, we investigate different methods of experience sharing that can lead to accelerated learning.

The proposed model is validated in two different reinforcement learning settings, a classical control and a navigation problem. The results shows that MACES is able to reduce in over a half the number of episodes required to complete a task through cooperation of only two agents, compared to a single agent baseline. The model is applicable to deep reinforcement learning agents.

**Keywords:** reinforcement learning, deep reinforcement learning, multiagent reinforcement learning, transfer learning, knowledge sharing

*"Your memories are the first step to consciousness. How can you learn from your mistakes if you can't remember them?"*

*Bernard Lowe in Westworld*

# Contents


**1 Introduction** **1**

1.1 Problems . . . 2

1.2 Objectives . . . 3

1.3 Document outline . . . 3

**2 Reinforcement Learning** **4**

2.1 Machine learning . . . 4

2.2 Markov decision process . . . 6

2.3 Q-learning . . . 8

2.4 Exploration vs exploitation . . . 10

2.5 State discretization . . . 12

2.6 Function approximation . . . 13

**3 Deep Reinforcement Learning** **15**

3.1 Deep learning . . . 15

3.2 Deep Q-network . . . 17

3.3 Experience replay . . . 20

3.4 Prioritized experience replay . . . 22

**4 Multiagent Settings** **24**

4.1 Markov games . . . 24

4.2 Learning goals . . . 26

4.3 Applications . . . 27

4.4 Knowledge sharing . . . 29

**5 Proposal** **32**

5.1 Experience sharing methods . . . 34

5.1.1 Naive ES . . . 34

5.1.2 Focused ES . . . 35

5.1.3 Prioritized ES . . . 36

5.1.4 Prioritized Focused ES . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.2 Baseline algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.3 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

**6 Empirical Validation** **42**
6.1 Experimental procedures . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
6.2 OpenAI: Cart Pole . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
6.2.1 Environment description . . . . . . . . . . . . . . . . . . . . . . . . 43
6.2.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
6.3 Microsoft: Marlo . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
6.3.1 Environment description . . . . . . . . . . . . . . . . . . . . . . . . 50
6.3.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

**7 Related Work** **55**
7.1 Sharing action advice . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
7.2 Sharing with heterogeneity . . . . . . . . . . . . . . . . . . . . . . . . . . 57
7.3 Increasing buffer diversity . . . . . . . . . . . . . . . . . . . . . . . . . . 59

**8 Conclusion** **62**

**References** **64**

**A Hyperparameters - Cart Pole** **69**

**B Hyperparameters - Marlo** **70**

# List of Figures


2.1 Classical diagram of a RL loop. Reproduced from Sutton and Barto (2018). 6
2.2 Representation of a Markov decision process. Reproduced from Sutton et al. (1999). . . . . 7
2.3 Example of Q-function values after 1000 iterations of Q-learning. Reproduced from Abbell and Klein (2014). . . . . 10
2.4 Graphical representation of coarse coding, with different levels of generalization. Reproduced from Sutton and Barto (2018). . . . . 13
2.5 Tile coding. Reproduced from Sutton and Barto (2018). . . . . 13

3.1 DNN representation with two hidden layers. Reproduced from LeCun et al. (2015). . . . . 16
3.2 Action-value function approximated by neural networks, in a DQN application to Atari 2600 games. Reproduced from Mnih et al. (2015). . . . . 18
3.3 Agent and environment interaction in DQN. Reproduced from Mnih (2017). 19
3.4 Experience replay implemented in DQN algorithm. Reproduced from Nair et al. (2015). . . . . 22

4.1 Representation of a Markov Game. The environment responds to a joint action. The new state and reward obtained is particular to each agent. Reproduced from Chincoli and Liotta (2018). . . . . 25
4.2 Soccer standard platform league match in RoboCup (2015). . . . . 28

5.1 Experience sharing architecture showing cooperation between two agents. The environment, replay buffer and inbox (shown in color) are unique for each agent, and the requests board (shown in gray) is shared among them. It can be extended to any number of agents. . . . . 33
5.2 Average pooling performed on a 4x4 input with a 2x2 kernel. The output result is the 2x2 matrix shown in the right. Reproduced from ReNom (2014). 36
5.3 Schematics of the Focused ES implementation . . . . . 37
5.4 Convolutional neural network architecture. . . . . 39

6.1 OpenAI CartPole environment . . . . . . . . . . . . . . . . . . . . . . . . 44
6.2 Comparison of single agent DQN with multiagent DQN with Naive ES and Focused ES. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
6.3 Comparison of single agent DQN-PR with multiagent DQN-PR with Prioritized ES and Prioritized Focused ES. . . . . . . . . . . . . . . . . . . . . 46
6.4 Comparison of multiagent variants DQN with Focused ES and DQN-PR with Prioritized Focused ES. . . . . . . . . . . . . . . . . . . . . . . . . 47
6.5 Episode reward evolution in DQN. . . . . . . . . . . . . . . . . . . . . . 48
6.6 Episode reward evolution in DQN-PR. . . . . . . . . . . . . . . . . . . . 48
6.7 Boxplots showing the distribution of number of episodes to complete task with different numbers of concurrently learning agents. . . . . . . . . . . 49
6.8 Density estimations for different number of tiles used on tile coding applied to Focused ES. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
6.9 Samples of the state perceived by the agent in Marlo environment . . . . . 51
6.10 Comparison of single agent DQN with multiagent DQN with Naive ES and Focused ES in Marlo environment. . . . . . . . . . . . . . . . . . . . . . 52
6.11 Comparison of single agent DQN-PR with multiagent DQN-PR with Prioritized ES and Prioritized Focused ES in Marlo environment . . . . . . . 54

7.1 Architeture showing how subordinate agents are grouped by the supervisor according to the calculated context features. Reproduced from Garant et al. (2017). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
7.2 Gorilla Framework for distributed learning with multiple learners and actors with a single replay memory and parameter server. Reproduced from Nair et al. (2015). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
7.3 Apex Framework for distributed learning with multiple actors, a single learner and a single replay buffer. Reproduced from Horgan et al. (2018). . 60

# List of Algorithms

1 Q-Learning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2 Deep Q-Network . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3 Experience sharing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

# List of Tables


5.1 Convolutional neural network detailed layers. . . . . . . . . . . . . . . . . . . 39

6.1 Experiments results in Cart Pole environment. . . . . . . . . . . . . . . . . . 47
6.2 Q1, Q2 and Q3 ETC for Focused Experience Sharing between 1 to 10 agents. 49
6.3 Comparison of results using different number of tiles for discretization. . . 50
6.4 Experiments results in Marlo environment. . . . . . . . . . . . . . . . . . . . 53

7.1 Related work comparison (in the order they are presented). . . . . . . . . 61

A.1 Hyperparameters selected for Cart Pole environment experiments . . . . . 69

B.1 Hyperparameters selected for Marlo environment experiments . . . . . . . 70

# List of Symbols

$S$ Set of possible states in the environment

$s, s'$ States

$s_t$ State at time t

$A$ Set of actions available to the agent

$a, a'$ Actions

$a_t$ Action at time t

$\bar{a}$ Joint action in multiagent settings

$r(s, a)$ Reward function

$r, r'$ Rewards

$r_t$ Reward at time t

$\pi$ Policy

$V(s)$ Value function

$V_i(s)$ Value function at iteration i

$V^*(s)$ Optimal value function

$Q(s, a)$ Action-value function

$Q_i(s, a)$ Action-value function at iteration i

$Q^*(s, a)$ Optimal action-value function

$\gamma$ Discount rate

$\delta$ Temporal-difference error

$\alpha$ Learning rate

$\epsilon$ Exploration rate

$\theta$ Parameters of neural network

$\nabla_\theta$ Gradient of parameters of a neural network

$\mathcal{L}$ Loss function

$\tau$ Soft update rate of target network

$\mathcal{D}$ Replay buffer

$\kappa$ Size of experience sharing batch

$\rho$ Priority of a transition

$RB$ Requests board

$\mathcal{R}$ Request

$\mathcal{A}$ Agent

$U(\pi)$ Utility function

$H_t(a)$ Preference function at time t

$Pr\{X = x\}$ Probability that a random variable $X$ takes on the value $x$

$(s_t, a_t, r_t, s_{t+1})$ Experience at time t

# Chapter 1

# Introduction

The field of artificial intelligence was born from an ambition to create machines as intelligent as the human species. Since the influential Darthmouth workshop in 1956, research into artificial intelligence has been driven by a desire to replace traditional programming, which requires step by step instructions for every task performed by a machine, for a more flexible scheme where a machine can learn by itself the optimal approach to reach an objective defined by its human creator (Russell and Norvig, 2009).

Reinforcement Learning (RL) is a learning mechanism where autonomous agents, through trial and error, attempt to learn the best action to take given a perceived state in an environment. It is part of the larger field of Machine Learning (ML), and it mainly differs from other ML approaches by its online nature. Instead of learning by analyzing a previously collected dataset, an agent learns through its interaction with the environment (Sutton and Barto, 2018).

The idea of sharing experience between cooperative agents emerges naturally from our view of how humans learn. While learning from experience, we also exchange knowledge with peers and teachers to accelerate learning, so learning involves as much information transfer as it involves discovery by trial-and-error. Our evolution as a species is tightly linked to the ability of sharing learned knowledge with each other, either instantaneously through verbal communications, or throughout generations by written culture. Experience Sharing (ES) between autonomous and independent agents could be paramount to replicate how humans learn, and greatly improve learning efficiency in ML methods applied to cooperative multiagent settings.

In this work, we explore the ES approach between homogeneous agents in the same environment. The MultiAgent Cooperative Experience Sharing (MACES) model is proposed, which comprises several methods of ES. First, we investigate the premise that sharing experiences alone is enough to increase learning performance of two or more cooperative agents. Following it, we propose a method which limits sharing to only expe-

riences which are novel to the learner agent. While random ES between two agents shows no improvement over single agent learning, the Focused ES method shows significant improvement over the baseline, reducing the number of episodes required to complete a task in over a half.

## 1.1 Problems

RL has many applications in industry, specially in problemas related to optimization, control and navigation. Its recent combination with deep learning reignited the field, and its core concepts are key to the latest developments in the field of ML (Fernandez and Mahlmann, 2018; Silver et al., 2016; Vinyals et al., 2017).

However, RL is considered difficult to implement, even for researchers in the field, as noted in Henderson et al. (2018a). It also requires more data to learn compared to other machine learning algorithms. Hence successful RL implementations are mainly seen in environments that can be simulated to generate large amounts of data, like robotics and games. A major challenge for the field, therefore, is to improve sample efficiency of existing RL algorithms, enabling an agent to learn more from less data.

This is the problem we address. We seek to accelerate learning in existing RL algorithms. Specifically, we look at settings where an agent can benefit from cooperation with other agents concurrently learning the same task. Learning performance is measured in number of episodes required to complete the task (ETC). Reducing ETC results in improvement in sample efficiency, and the benefits can be extended to a large number of applications currently implementing RL in cooperative multiagent scenarios.

As a proposed solution, we explore how to effectively implement experience sharing between cooperative agents. Sharing experiences in RL agents was first investigated in Tan (1993) and Whitehead (1991). One of the main issues in learning by trial-and-error is that it relies on the agent's luck in first achieving the goal by chance, which could be overcome by learning a policy directly from external experts (Lin, 1992). If cooperation is done intelligently, each agent can benefit from other agents' instantaneous information, episodic experience, or learned knowledge (Tan, 1993).

Our research situates in the field of Multiagent Reinforcement Learning (MARL), and the model proposed applies to RL algorithms that make use of neural network as function approximators, a class of algorithms is known as Deep Reinforcement Learning (DRL). Improving learning with cooperation, if successful, can be extended to several practical applications in RL, specially robotics and navigation problems, where it is common to have several potentially cooperative agents performing the same task.

## 1.2 Objectives

Our goal is to evaluate ES methods between cooperative RL agents. We want to evaluate if ES can be an effective tool to accelerate learning in cooperative multiagent reinforcement learning settings. The metric used as benchmark for evaluation purposes is number of episodes required to complete a task, or ETC.

The main goal can be broken down into two subgoals: (i) show two or more agents cooperating through ES can learn faster than a single agent; (ii) propose enhanced methods of ES that can lead to faster learning. Accomplishing these subgoals will result in the following contributions:

- Study and propose enhanced methods of ES that can accelerate learning;
- Propose a general model for ES among cooperative agents;
- Implement the proposed ES model and its methods;
- Validate the proposal in two distinct RL scenarios: classical control, and navigation.

## 1.3 Document outline

This document is organized as follows.

- **Chapter 2 - Reinforcement Learning** introduces the basics concepts of ML and RL;
- **Chapter 3 - Deep Reinforcement Learning** discusses neural networks and how they can be combined with RL;
- **Chapter 4 - Multiagent Settings** describes the extension of RL to scenarios with more than one agent;
- **Chapter 5 - Proposal** presents the proposed model MACES and methods of ES;
- **Chapter 6 - Empirical Validation** presents experimentation results and discusses the results achieved;
- **Chapter 7 - Related Work** overviews the literature related to the proposal;
- **Chapter 8 - Conclusion** discusses limitations, possible applications and future work.

# Chapter 2

# Reinforcement Learning

This chapter discusses RL. We start with a general introduction to the more broad category of ML, and lead on to investigate RL in more details.

## 2.1 Machine learning

‘

Machine Learning, a subarea of Artificial Intelligence, is a term used to describe algorithms used to learn patterns from data. It is commonly divided into three different subclasses: Supervised Learning (SL), Unsupervised Learning (UL) and RL.

The first category, which has gained a lot of traction recently with online competitions and industrial applications, is called SL. It is defined by Russell and Norvig (2009) as follows:

> The task of supervised learning is this: given a training set of N example input–output pairs $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$, where each $y_j$ was generated by an unknown function $y = f(x)$, discover a function $h(x)$ that approximates the true function $f(x)$. Here $x$ and $y$ can be any value; they need not be numbers. The function $h$ is a hypothesis.

SL algorithms learn to identify a pattern in a given labeled dataset. The algorithm encodes a model, which represents the pattern. A common SL model is composed of a set of parameters, with each parameter having an associated weight. The learning task then becomes an optimization task, which can be described as finding the optimal combination of weights that makes the model a better predictor of new observations.

This problem is also called credit assignment path (Schmidhuber, 2015), due to the fact the optimization algorithm has to discover how each parameter contributed to the correct or incorrect classification of a new observation, and adjust its weights accordingly in order to improve its chance to correctly predict the next round. The paramount premise in SL is

that the model has access to large amount of labeled data, since during the training stage the model requires knowing whether an item has been correctly or incorrectly classified.

In contrast, methods of the second category, UL, do not require a labeled dataset. They learn patterns only through analysis of the data itself. As defined by Russell and Norvig (2009):

> In unsupervised learning the agent learns patterns in the input even though no explicit feedback is supplied. The most common unsupervised learning task is clustering: detecting potentially useful clusters of input examples.

Apart from clustering, another common application of UL are dimensionality reduction techniques, such as principal and independent component analysis. UL has been relevant to the evolution of ML algorithms since 1970. Today, its largest contribution is as an aid to SL algorithms, being used to preprocess datasets, reduce dimensions, select relevant features for maximum entropy, and pre-initialize weights and hyperparameters for SL models. The first breakthrough achieved by Deep Neural Network (DNN)s in the 2000's decade was attributed to the use of AutoEncoders, an UL neural network, to pre-train deep feedforward neural networks (Schmidhuber, 2015).

The third subclass of ML algorithms is called Reinforcement Learning. Russell and Norvig (2009) defines as:

> In reinforcement learning the agent learns from a series of reinforcements—rewards or punishments. For example, the lack of a tip at the end of the journey gives the taxi agent an indication that it did something wrong. The two points for a win at the end of a chess game tells the agent it did something right. It is up to the agent to decide which of the actions prior to the reinforcement were most responsible for it.

RL deals with the problem of online learning; instead of learning a pattern through large amounts of data previously made available, RL algorithms learn by direct interaction with the environment, using feedback perceived at each step to adjust its action-selection policy. RL algorithms and its history are discussed in details in this chapter.

The distinction between these three subclasses of ML algorithms is becoming more blurry in the current state of the art research. UL models are used to enhance SL algorithms, with improvements in performance or running time. More recently, SL and UL have been combined with RL with astonishing results in different domains, setting the path for a new subclass of algorithms, DRL (Schmidhuber, 2015).

## 2.2 Markov decision process

Reinforcement learning can be summarized as learning through rewards. In a more detailed description, RL concerns an agent learning in a given environment. The agent perceives the world through sensors, and changes it through its actions. Each action taken by the agent affects the environment, that may output a reward (Sutton and Barto, 2018), as seen in Figure 2.1. The learning process based on reward is inspired on the RL theory developed in psychology to explain animal behavior (Ferster and Skinner, 1957).

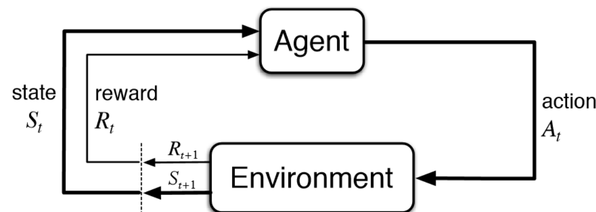


Figure 2.1: Classical diagram of a RL loop. Reproduced from Sutton and Barto (2018).

The RL problem is commonly modeled as a Markov Decision Process (MDP), formalized by a set of states $S$, a set of actions $A$, a transition function

$$p(s' \mid s, a) = Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}, \tag{2.1}$$

and a reward function

$$r(s, a) = \mathbb{E}[r_{t+1} \mid s_t = s, a_t = a], \tag{2.2}$$

where $s_t$ is the current state at time $t$, $a_t$ is the action taken at time $a_t$, and $s_{t+1}$ is the state at time $t+1$. The agent moves from one state to another through its actions. The transition probability function determines which next state $s'$ will the agent arrive after taking action $a$. After arriving at the new state, the agent receives a reward, which can be null, positive or negative (Sutton and Barto, 2018). This cycle is represented in Figure 2.2.

The goal of an RL agent in this MDP setting is to learn an optimal policy $\pi$, which leads to the maximum reward possible. Policy is a function that determines which action the agents needs to take given the perceived state. If we consider a finite amount of time $n$, every sequence of actions from the agent from time 0 to time $n$ is considered an episode. The agent thrives to maximize not only local reward of a step, but the total reward for an episode. The total reward can either be spread upon intermediate states or concentrated in the final state, introducing the problem of learning an optimal policy in a delayed rewards setting.

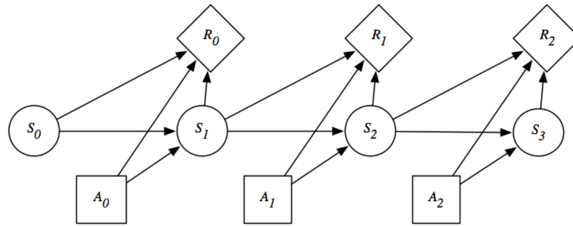


Figure 2.2: Representation of a Markov decision process. Reproduced from Sutton et al. (1999).

An important classification of RL is between model-based and model-free algorithms. The first relies on the premise that, given the full knowledge of the environment, including the transition or the reward functions, one can analytically solve for the optimal policy. If the agent has access to complete knowledge of the environment, then learning is no longer required; planning algorithms can be used to solve the MDP. State-of-the-art model-based algorithms involve some sort of model estimation to increase sample efficiency in learning algorithms by combining it with planning.

Model-free, on the other hand, considers an agent can only sample transitions from the environment by directly experiencing it. In model-free algorithms, solving for an MDP can be either done by directly optimizing the policy, a function which maps state to actions and tells the agent what to do in a given situation, or by learning a function that attributes a utility value to each state. We call the first approach policy-based and the last value-based algorithms. By estimating the utility value of each state, a value-based algorithm can derive a policy by selecting the action which yields the greatest utility. This approach has been more explored in the RL literature due to its property of sample efficiency, leading to the popular Temporal-difference (TD) learning algorithms SARSA, Q-Learning, and its variants (Sutton and Barto, 2018).

A value-based algorithm attempts to calculate the optimal value function $V^*$. The value of a state is the maximum expected cumulative reward $r$ achievable from that state $s$, given a policy $\pi$. It can be formalized by:

$$V^*(s) = \max_{\pi} \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, \pi\right], \tag{2.3}$$

.

where $\gamma$ is a discount factor between 0 and 1 applied to discount future rewards. We can approach this problem in an iterative fashion. Bellman's principle of optimality affirms that:

> An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision (Bellman, 1957).

The principle of optimality allows us to consider each decision separately. So we can solve for it iteratively making use of dynamic programming techniques, by breaking the optimization problem into a sequence of simpler problems. The equation can be rewritten recursively as the expected reward achieved in the state plus the expected reward achieved from the next state:

$$V_{i+1}(s) = \mathbb{E}\left[r + \gamma V_i(s') \mid s\right], \tag{2.4}$$

where $s'$ is the next state. We can also extend it to the state-action formulation, where $Q$ is the action-value function:

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_a Q_i(s',a) \mid s,a\right]. \tag{2.5}$$

Using the Bellman equation allow us to solve for the value function with repeated small updates. This gives us the convenient property that, by taking fewer steps than required to achieve the optimal value function, we can trade sample efficiency for a less accurate estimate of the true value of a state. More often, a loosely estimation of the true value function may be enough to arrive at the optimal policy. This formulation is the basis of TD-learning algorithm Q-Learning, which we will investigate next.

Policy-based algorithms, on the other hand, rely on direct search in the policy space to find the optimal policy. Optimization methods can include gradient-based methods, such as gradient ascent, genetic algorithms or Monte Carlo techniques based on repeated sampling. Policy-based methods has recently resurfaced with the family of policy gradient algorithms which use complex non-linear models to model the policy and optimization tricks to speed up learning. A third class of algorithms called Actor-Critic, combines value-based and policy-based approaches, using the gradient from TD-error based optimization to direct the policy search (Sutton and Barto, 2018).

## 2.3 Q-learning

Q-Learning is a model-free algorithm which belongs to a class of RL methods known as Temporal-difference (TD) learning, where the value function is updated by the difference between the perceived reward plus expected reward from the future states and the expected reward from current state, which is known as the TD-error.

In Q-Learning, the agent attempts to learn an optimal action-value function $Q^*(s,a)$, which maps a state and action to a utility value. The $Q$ function will be used to derive a behavior policy $\pi$. During a pre-defined number of episodes ($i$), at each time $t$ the agent experiences the world by choosing an action $a$ from state $s$, reaching the next state

$s_{t+1}$ and perceiving a reward $r$. The reward obtained is used to update its action-value function $Q$. It continues to interact with the environment until the episode ends, when it moves back to the initial state and starts a new episode. This loop is represented in Algorithm 1 (Sutton and Barto, 2018; Watkins and Dayan, 1992).

**Algorithm 1** Q-Learning

1: **for** $i \leftarrow 1$ to number of episodes **do**
2: $t \leftarrow 0$
3: **while** $s_t$ is not a terminal state **do**
4: $t \leftarrow t + 1$
5: choose an action $a_t$ according to policy $\pi$
6: execute $a_t$
7: update $s_t$ to new state observed
8: update $Q$-function
9: **end while**
10: **end for**

In Q-Learning, the policy derives from the Q-function. The most simple policy is the greedy policy: given a state, always select the action that leads to highest estimated utility. Most policies blend greedy actions with exploratory behavior, specially in the earlier episodes, to allow the agent to improve its Q-function to yield better predictions of state-action values. The update rule of the Q-function is given by:

$$\delta = r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \tag{2.6}$$

$$Q_{i+1}(s_t, a_t) = Q_i(s_t, a_t) + \alpha\delta. \tag{2.7}$$

Equation 2.6 represents the TD error $\delta$. The variable $\alpha$ is a step-size learning rate, which allows for gradual adjustment towards the target, and $\gamma$ is an exponentially decreasing discount factor attributed to future rewards. The discount factor is required to extend Q-learning to infinite horizon. Non-discounted future rewards can lead to policies where the agents will have no incentive to choose an action over another, since both will lead to the same reward in the long run. The discount factor can be seen an incentive for the agent to achieve a reward as early as possible in the episode.

Q-Learning is guaranteed to converge to optimal policies in stochastic scenarios (Watkins and Dayan, 1992). In non deterministic environments, it is easy to see that if we continuously loop through the problem, we will eventually reach a state with high reward. We only need to reach the high reward state once, which will trigger an update of the Q-function, so the following iterations will choose the action with highest reward in the next loop.

Update of the Q-function is done through iterative update of a table which maps state and action to utility values, as exemplified in Figure 2.3. In the left side, we can see the action-value for each state-action pair after only 5 episodes. Through exploration, the agent has found the terminal state with reward of 1. The reward of the terminal state is propagated to the previous state at each iteration, using Equation 2.7. After 1000 episodes, the action-value for each state is closer to the optimal action-value function, $Q^*$, as shown in the right. $Q^*$ can be used to derive the optimal policy.

Such approach is not practical in high dimensional state-action spaces or feasible in spaces represented by continuous variables. A solution is to approximate the Q-function using linear or non-linear models, which we will see in Section 2.6.

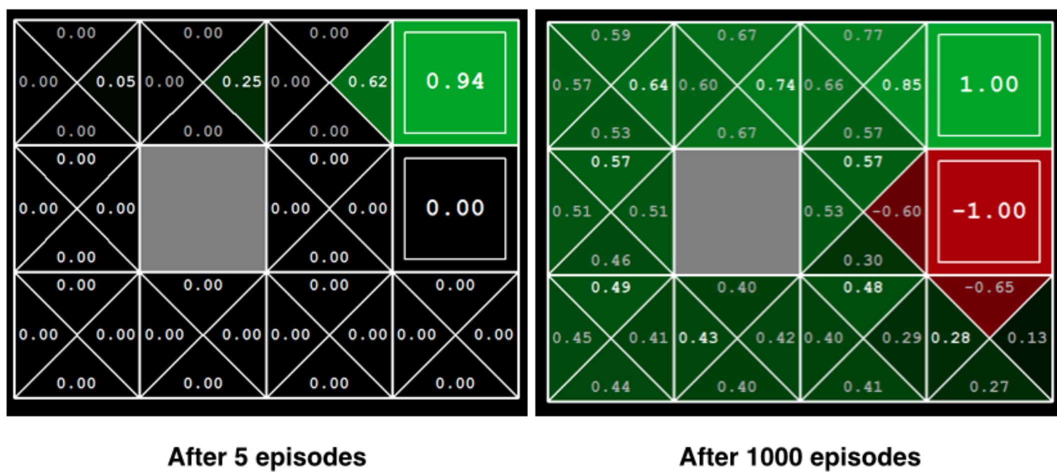


Figure 2.3: Example of Q-function values after 1000 iterations of Q-learning. Reproduced from Abbell and Klein (2014).

## 2.4 Exploration vs exploitation

In supervised learning, we commonly assume an underlying data distribution function, and all samples used for training and test needs to be identically independent samples drawn from that distribution. No such assumption can be made for RL, where an agent learns through interaction with the environment, and the experiences used for learning are therefore dependent on the actions taken by the agent. An agent who explores a fixed policy will only learn from transition samples related to a small region of the state space which the policy leads to, and thus will fail to find the optimal policy. This is known as exploitation. On the other hand, an agent that always choose its actions randomly, exploring the environment, will be stuck in a suboptimal policy.

This is known as the exploration-exploitation dilemma, and is one of the key research fields in RL. In deterministic policies, a common approach to address this dilemma is known as $\alpha$-greedy strategy. At each step, an agent greedily follows the optimal action in a learned policy $\pi$ with $1-\epsilon$ probability, and randomly chooses amongst the available actions with probability $\epsilon$:

$$a = \begin{cases} \arg\max_a Q(s,a) \text{ with probability } 1-\epsilon \\ \text{a random action with probability } \epsilon \end{cases} \tag{2.8}$$

The variable $\epsilon$ is a tunable hyperparameter, and can either be fixed at the start of the learning process or decayed over time. Convergence guarantees are only theoretically available for agents whose starts with a high degree of exploration and decay over time, moving from a full exploration to a full exploitation policy.

Possible improvements to the $\epsilon$-greedy strategy includes selecting amongst non-optimal actions according to their potential for being optimal, taking into account how close the Q-function estimates are to the value of the optimal action and the uncertainty in these estimates. This method is called upper-confidence bound action selection (Sutton and Barto, 2018). Exploration can also be forced by setting optimistic initial values for every state-action pair, forcing the agent to visit at least once all state-action pairs, as long as the initial value defined is equal or higher the maximum reward obtainable for any state.

Stochastic policies are a more natural approach to handle exploration. If the probability for each action $a$ at a state $s$ is attributed equal weights at the beginning of the learning process, the agent's initial stochastic policy will be similar to a deterministic policy with $\epsilon$ set to 1. As the agents learn, the probabilities are gradually adjusted to favor actions which are more likely to lead to positive rewards. As long as all actions probabilities at every possible state $s_i$ are $< 1$, the agent will continue exploring the environment at some level. Continuous action spaces where action is represented by a scalar or a vector of scalars are a natural fit for stochastic policies.

Deterministic action, however, can also be represented stochastic policies by assigning a numerical preference for each action, $H_t(a)$, and using a softmax distribution (also known as Gibbs distribution or Boltzmann distribution) over the preferences to determine the probability of selecting the action, as follows:

$$Pr\{a_t = a\} = \frac{e^{H_t(a)}}{\sum_{a'=1}^{|A|} e^{H_t(a')}} = \pi_t(a) \tag{2.9}$$

Preferences are initialized to the same constant, and after each step they are updated according to the reward received upon taking an action, with

$$H_{t+1}(a_t) = H_t(a_t) + \alpha(r_t - \bar{r_t})(1 - \pi_t(a_t)) \tag{2.10}$$

for $a = a_t$ and

$$H_{t+1}(a) = H_t(a) + \alpha(r_t - \bar{r_t})\pi_t(a) \tag{2.11}$$

for all $a \neq a_t$, where $\alpha > 0$ is a step size parameter and $\bar{r_t} \in \mathbb{R}$ is the average of all rewards up through and including time $t$. This method is popularly known as Boltzmann exploration (Derthick, 1984; Sutton and Barto, 2018).

## 2.5 State discretization

State and action spaces can be represented by a set of discrete or continuous variables. If all variables which represent the state-action space are discrete, and the number of possible combinations is low, the state-action value function can be represented by a table. Otherwise, the function needs to be approximated by a model.

Discretizing a space is the simplest form of function approximation. In RL, discretization can be applied for the state space alone, which is the most common application, for action space alone, or for the state-action space combined.

State aggregation is implemented by grouping states together, with one estimated value per group. The estimated value attributed to a state is then the estimated value of the group it belongs to. When updating the state value function, only the values pertaining to the group the state belongs to are updated (Sutton and Barto, 2018).

State aggregation introduces hard boundaries which can lead to artifacts in function approximation, specially when too many state values lay close to the boundaries. An alternative, known as coarse coding, is to encode the state in a number of features with different receptive fields. This can be visualized as overlapping circles covering the entire state space, as shown in Figure 2.4. The state is represented as a binary vector, with 1 for every circle where it is in and 0 where it is not. This allows for greater generalization, which can be controlled by the radius of the circles, and prevents the artifacts created by the boundary cases in regular state aggregation.

We can extend discretization to multi-dimensional continuous spaces, a method named tile coding (Sutton, 1996). Multiple groupings are used, known as tilings, and each one is slightly offset in a direction to represent a different grouping of the state space. As in coarse coding, the tilings overlap. Wherein state aggregation each variable is converted

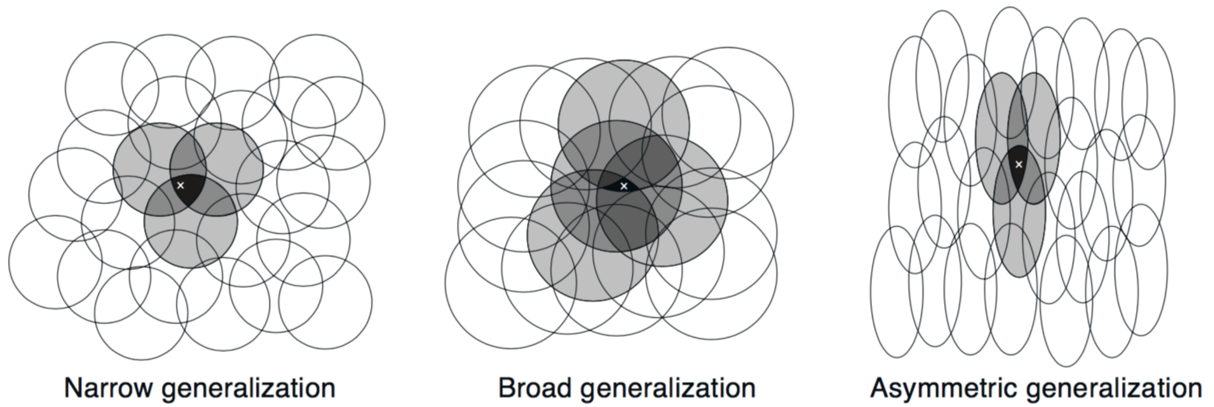


Figure 2.4: Graphical representation of coarse coding, with different levels of generalization. Reproduced from Sutton and Barto (2018).

into a category represented by a single scalar, in tile coding each variable is encoded in a category represented by a vector of scalars. This scheme is represented in Figure 2.5.

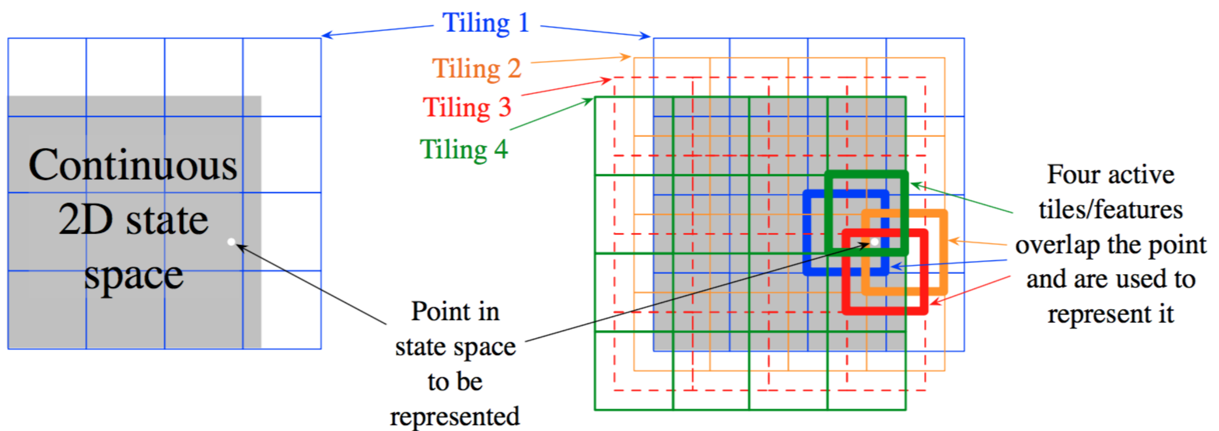


Figure 2.5: Tile coding. Reproduced from Sutton and Barto (2018).

## 2.6 Function approximation

Approximating the value function by a model allows handling of continuous state spaces, complex state spaces and more importantly, increases the model generalization, an important requirement for efficient RL. To be able to learn an optimal policy for $m$ possible spaces by being exposed to $n$ transitions, where $n < m$, an agent needs to be able to extend a policy learned for a specific state to nearby or similar states.

Generalization is a topic studied under SL, which can be successfully applied to the RL setting. Functions learned in RL can be approximated by a SL model. In model-based algorithms, the transition and reward function, which combined represent the environ-

ment's dynamics, are candidates for function approximation. In model-free algorithms, the value or action-value function and the policy can in theory be approximated.

Discretization, presented in section 2.5, is one of the methods used to approximate a function. Linear models have also been used in RL with significant success for simple problems. An action-value function approximated by a linear model has the form of:

$$Q(s, a, \theta) = \sum_{i=1}^{d} \theta_i \phi_i(s, a) \tag{2.12}$$

where $\theta$ is the set of parameters of the linear model and $\phi$ represents the features of the state-action space. However, the hypothesis space of possible solutions of a linear model is the set of all linear functions of its input (Goodfellow et al., 2016). We can generalize linear models to expand the space of solutions it can choose from, transforming the input by constructing features that allow it to approximate non-linear functions. Features commonly used are polynomials, Fourier basis and radial basis functions, as well as the features introduced by the discretization methods discussed (Sutton and Barto, 2018).

A more direct solution would be to use a non-linear function approximator. Neural networks have the interesting property of universal function approximators, and for this reason have been studied as viable function approximators for as long as RL algorithms have been studied. In a study by Lin (1992), neural networks are used to approximate the action-value function in the Q-learning algorithm. The same work introduces methods and heuristics to safely adapt neural networks for the RL context, including the introduction of an experience buffer to ensure transitions are used more than once in the network optimization process before they are discarded. Tesauro (1995) used a similar approach to construct a world-class backgammon playing algorithm, a significant milestone in the artificial intelligence research field development.

The features used by Tesauro as input to its algorithm were manually handcrafted for the specific problem. It took more than 20 years to successfully extend this approach to more complex problems with a more general algorithm, benefiting from the advances in representational learning by the emerging field of deep learning. This is what we will cover in Chapter 3.

# Chapter 3

# Deep Reinforcement Learning

In this chapter, we give a brief review of the field of deep learning and how it relates to RL, introducing the class of algorithms known as Deep Reinforcement Learning (DRL).

## 3.1 Deep learning

In the past years, the artificial intelligence research community has seen a lot of hype around a class of ML methods called deep learning. Although the naming is new, coined around 2006 (Schmidhuber, 2015), the algorithm which originated it dates back to 1940 when McCulloch and Pitts (1943) first transcribed the neuron behavior. The described behavior inspired an artificial system that could model complex non-linear dynamics, named Artificial Neural Network (ANN).

The reason of this hype are the recent results achieved by DNN algorithms in ML contests. DNNs have started winning ML competitions since 2003, starting with NIPS 2003 Feature Selection Challenge, a well-known contest with a secret testing set. More recently, with the advent of Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) models, deep learning algorithms have achieved far better performance in a wide variety of secret and open test set contests, including MNIST from 2009 to current date, and ImageNet from 2012 to 2018 (Krizhevsky et al., 2012; Salakhutdinov and Hinton, 2009).

Apart from contests, DNNs have also made a grand entrance into the corporate world, proving a valuable algorithm to solve many existing problems such as cancer diagnosis, object detection and localization and video classification. The big advantage of DNNs over traditional ML methods is the lack of preprocessing required in order to fit a dataset to the algorithm. CNNs are able to automatically learn features for classification, exempting the practitioner of the expensive task of feature engineering and image preprocessing. RNNs can learn patterns in sequence of data, capturing long and short temporal patterns,

without data compression techniques commonly used before to handle time series data (Pascanu et al., 2013).

The term deep learning comprises neural networks containing more than one hidden layer between the input and output nodes. A sample architecture is shown in Figure 3.1. Multilayer perceptrons are considered to be the first DNNs, and have been widely used in the SL community around the 80s and 90s (LeCun et al., 2015). But until late 2000s, other SL algorithms have been used with greater success in terms of performance and computational cost, including Support Vector Machines (Hearst et al., 1998), a big hit in early 2000s, and ensembles of simple classifiers such as Random Forest and Gradient Boosting (Breiman, 2001; Friedman, 2001).

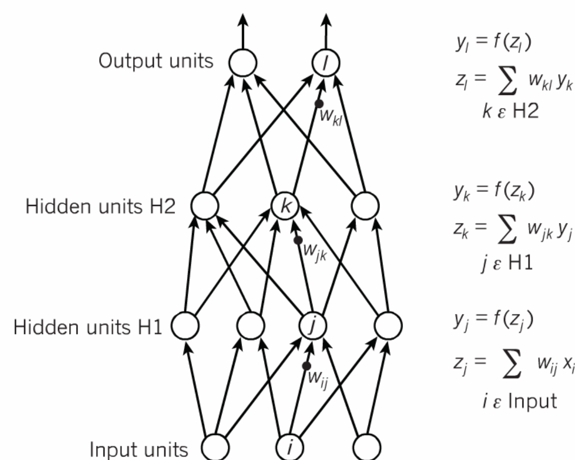


Figure 3.1: DNN representation with two hidden layers. Reproduced from LeCun et al. (2015).

DNNs include not only supervised, but also unsupervised learning models. The term DNN has emerged with the success of stack of autoencoders, an UL method used to compress representations, and deep belief networks, which belong to the class of generative neural networks (Schmidhuber, 2015). Two variations have attracted a lot of interest from the research community and industry.

The first is CNN, largely used in object detection and classification tasks. CNNs are comprised of several convolutional layers, which act as feature selectors, mimicking the workings of the visual cortex. Each layer learns to recognize features such as edges, simple shapes, or changes in contrast, features that can be relevant for the classification task. Recent improvements to CNNs includes the use of max-pooling layers and dropout as regularization techniques. Several architectures and algorithm variations of CNN are currently in use. One that is drawing attention lately are Fully Convolutional Networks, used for semantic segmentation or pixel by pixel classification (Long et al., 2015).

The other relevant model which has become dominant recently are Recurrent Neural Networks. These architectures are able to preserve the temporal information in sequential datasets, by using information from previous observations while learning the weights for the current observation. The basis for RNNs is a memory cell with a forget and a keep gates. Both gates have weights that can be trained, so the network learns whether an information needs to be forgotten or is still relevant to understand the pattern of the current observation.

As with CNNs, several RNNs variations are being deployed. The most widely used are Long Short Term Memory Networks, which uses a complex memory cell by the same name (Hochreiter and Schmidhuber, 1997). RNNs work well in sequential datasets or in any domain in which the temporal dimension is relevant, which may include audio preprocessing, natural language processing, and object detection in videos.

Human sensory input streams are sequential, so RNNs could in theory be extended to any pattern detection task performed by humans. Vision, for example, although perceived as static, is also sequential, as vision works through rapid eye movement called saccades. There is a large body of research concerning attention models based on the workings of the human eye, transforming image to sequential data that can be decoded by RNNs (Mnih et al., 2014).

## 3.2 Deep Q-network

Recent research has shown the human mind is able to combine feature extraction and image classification techniques with learning through reinforcement (Yamins and DiCarlo, 2016). Several attempts have been made to combine these two classes of algorithms, mostly by using ANNs to approximate the value or action-value function. However, past experiments have not shown good results, showing ANNs overfitted or diverged when used as nonlinear function approximators in RL problems (Schmidhuber, 2015).

In 2015, the DeepMind research group was able to successfully combine both and published an algorithm known as Deep Q-Network (DQN) (Mnih et al., 2015), achieving a major breakthrough in solving sequential decision problems. A representation of the neural network used as Q-function approximator is shown in Figure 3.2. Only a few months later, several features from DQN were used to create the AlphaGo model that beat the Go world champion Lee Sedol in a 4-1 match (Silver et al., 2016).

DQN brings innovations that solve the unstable problems introduced in using neural networks as function approximators. DQN makes use of the Experience Replay (ER) technique, first introduced by Lin (1991), which consists in keeping history of past transitions, and randomly drawing from this transitions memory to update the network weights.

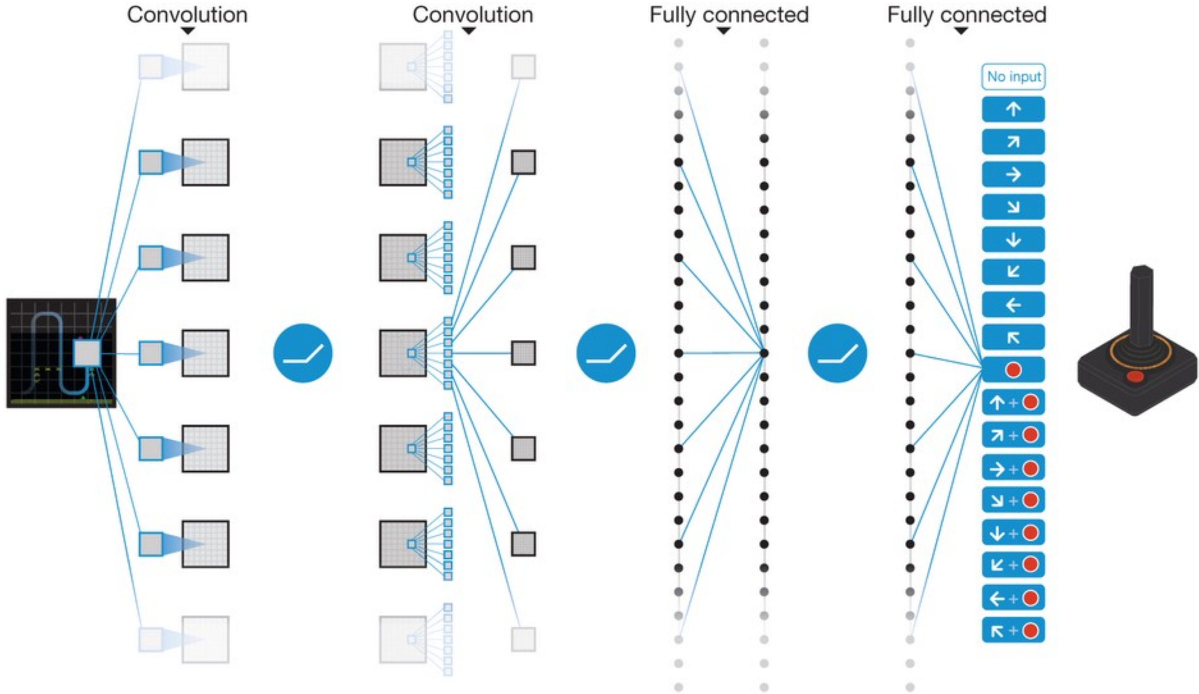


Figure 3.2: Action-value function approximated by neural networks, in a DQN application to Atari 2600 games. Reproduced from Mnih et al. (2015).

Several algorithms that combined DNN and RL followed the release of DQN (Sallab et al., 2017).

The DQN algorithm was devised as a DRL algorithm that could be extended to a wide range of competencies in a variety of tasks. Its initial implementation applied a CNN to represent the action-value function $Q(s, a)$, mapping a state, represented by an image, to a discrete set of actions.

Two key ideas were introduced to fix the stability issues found in previous attempts to combine RL and ANN. To account for overfitting to recent experiences, the agent does not update the action-value function at every new experience. Instead, it keeps a buffer of experiences, and at every step samples $\mathcal{D}^+$ random experiences from this buffer and use them to update the action-value function. A general scheme of DQN is shown in Figure 3.3.

The other relevant issue adressed by DQN is the moving target problem. The current value of the Q-function is used as part of the update of the Q-function itself. This procedure, known as bootstraping, creates a moving target problem for the learning function, since each action-value update changes the target as well. In DQN, this issue is fixed by implementing a second set of weights (which can be interpreted as a second network) that are only updated periodically, thereby reducing correlations with the target. The full pseudocode is shown in Algorithm 2, adapted from Mnih et al. (2015).

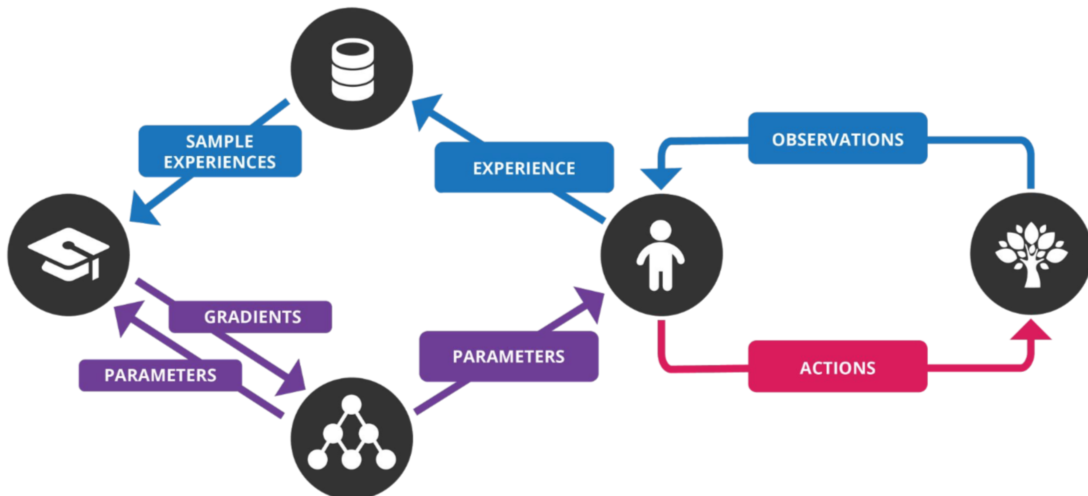


Figure 3.3: Agent and environment interaction in DQN. Reproduced from Mnih (2017).

**Algorithm 2** Deep Q-Network

1: **for** $i \leftarrow 1$ to number of episodes **do**
2: $\quad t \leftarrow 0$
3: $\quad$ **while** $s_t$ is not a terminal state **do**
4: $\quad\quad t \leftarrow t + 1$
5: $\quad\quad$ choose an action $a_t$ according to policy $\pi$
6: $\quad\quad$ execute $a_t$
7: $\quad\quad$ store transition $s_t, a_t, r_t, s_{t+1}$ in replay buffer
8: $\quad\quad$ update $s_t$ to new state observed
9: $\quad\quad$ sample batch of experiences $\mathcal{D}^+$ from replay buffer
10: $\quad\quad$ update Q-function
11: $\quad$ **end while**
12: **end for**

As in Q-Learning, the Q-function in DQN is updated according to the TD-error, which is calculated over the batch of experiences sampled from buffer instead of a single transition. The expected squared TD-error over the batch of experiences serves as the loss function of the neural network that approximates the Q-function. The parameters of the neural network are updated according to the TD-error based loss $L$, in the direction of the gradient of $\theta$, as follows:

$$\mathcal{L}_i(\theta_i) = \mathbb{E}_{s_t, a_t, s_{t+1}, r_t \sim \mathcal{D}} \left[ r_t + \gamma \max_a Q(s_{t+1}, a; \theta_i^-) - Q(s_t, a_t; \theta_i) \right]^2 \tag{3.1}$$

$$\theta_{i+1} = \theta_i - \alpha \nabla_\theta \mathcal{L}_i(\theta_i). \tag{3.2}$$

The learned network parameters are represented as $\theta$ and the target network as $\theta^-$. Every $n$ steps, the parameters of the target network are copied over the learned network.

An alternative, called soft update, is to modify $\theta$ every step with a small percentage of $\theta^-$. The step size update is defined as the hyperparameter $\tau$, giving the following update function:

$$\theta_i = (1 - \tau)\theta_i + \tau\theta_i^-. \tag{3.3}$$

DQN was successfully applied to the Arcade Learning Environment (ALE). Using a single instance of DQN, an agent learned to play 49 Atari 2600 games, achieving above human performance in almost all of the games. More recent updates to the DQN algorithm have increased the performance even further to include all Atari 2600 games (Hessel et al., 2018).

The current state of the art DQN based algorithm, named Double-DQN, proposes using the learned network to select the next action, and the target network to get the value for the next state-action (Van Hasselt et al., 2016). The modifications yields a new loss function:

$$a_{t+1} = \arg\max_a Q(s_{t+1}, a; \theta_i) \tag{3.4}$$

$$\mathcal{L}_i(\theta_i) = \mathbb{E}_{s_t, a_t, s_{t+1}, r_t \sim \mathcal{D}} \left[ r_t + \gamma Q(s_{t+1}, a_{t+1}; \theta_i^-) - Q(s_t, a_t); \theta_i) \right]^2. \tag{3.5}$$

Replacing the max operator in DQN for an argmax helps convergence by reducing the optimism bias introduced by the max operator in future reward estimation. Double-DQN has shown to ease the overestimation of Q-values and both speed up and stabilize training.

## 3.3 Experience replay

Experience Replay (ER) is first introduced by Lin (1991). The author motivates his work by describing self-improving by trial and error as hazardous in a hostile environment, and efforts to reduce learning time could therefore minimize the hazard. He notes that some experiences may be rare and are costly to obtain, so it does not make sense to discard the experiences obtained through RL after they are used only once.

Lin also introduces the notion of experiences as quadruples, composed of state the agent is currently in ($s_t$), the action it took ($a_t$), the reward it receives from the environment ($r_t$) and the next state the agent is transitioned to ($s_{t+1}$), or a $(s_t, a_t, r_t, s_{t+1})$ quadruple. The experiments conducted by Lin in 1991 and 1992 are prescient and in many ways close to the DQN methodology used in 2013 to jumpstart the field of DRL.

One of the issues addressed by Lin is the possible discrepancies between the behavior policy, followed at the time the experience was taken, and the learning policy, the

current active policy the agent is learning, Since Q-learning is an off-policy method, its convergence properties in a table implementation value function is not affected by these differences. However, using function approximators to approximate the action value function introduces limited degrees of freedom, and suddenly not all states and actions can be represented. Whenever backpropagation or related optimization methods modifies a value function with respect to one input state, it also affects the function with respect to many or possibly all input states.

To avoid divergence, Lin seeks to introduce recency in the algorithm. For that purpose the replay buffer is kept to a limited size, being constantly renewed by new experiences, and the experiences are always played in backwards manner using a recency factor to ensure the most recent experiences have a greater impact in the value network weights. The experiences also pass through a filter before they are used - if the probability of belonging to the current policy is less or equal a pre-defined threshold, they are discarded from the sample. This threshold is optimized as a hyperparameter, with ideal values ranging from 0.1 to 0.2. In more recent applications, the threshold is replaced by the importance sampling methodology, which weights experiences of a behavior policy $b$ by their probability of belonging to the learning policy $\pi$ (Hachiya et al., 2009; Sutton and Barto, 2018).

Another interesting feature of Lin's earlier work on ER, which adds to the described filter, is the screen test. Lin proposes that before an experience is replayed, the agent verifies its probability of choosing that action according to the current policy and Q-function. If the probability is higher than a given threshold (set initially as 99%), or lower than a second threshold (set initially to 0.01%), the experience is not replayed. Not passing the screen test implies the agent already knows the action is much better than others or much worse than the best one, and would not improve the Q-value function further. The procedure was validated empirically, and found that ER without the screen test was only beneficial in the beginning and actually harmful after a while. This feature has been dropped for more recent implementations of ER, including DQN.

Although most commonly applied to model-free algorithms, ER can essentially be seen as an approximation to model-based RL algorithms. Instead of using transitions to estimate a model that can generate future samples for offline learning, the agent keeps a buffer of those transitions and sample them randomly. The experience buffer acts as a non-parametric model of the environment. The similarity between these approaches have already been noted in Lin (1992), which compared Q-Learning with ER and Adaptive Heuristic Critic, a model-based method, and concluded that the first learned better. He follows it by remembering an estimated model of the environment could still be useful to perform conventional look-ahead search, but ER was a better option than relaxation

planning to extracting extra samples from the environment. This relationship has also been thoroughly explored in more recent research (Altahhan, 2018; Vanseijen and Sutton, 2015).

During the past two decades ER had been sporadically addressed in the literature, mainly as an additional methodology for data efficiency in complex domains, as seen in the works of Smart and Kaelbling (2000) and Kalyanakrishnan and Stone (2007). In Adam et al. (2012), the authors describe a general framework for ER and conduct a series of experiment focused specifically on ER with robust results in real and simulated applications. Shortly after the methodology was considered as key to the success of the DQN algorithm (Mnih et al., 2015) and following DRL variants, and the interaction between the replay memory buffer and the network can be seen in Figure 3.4.

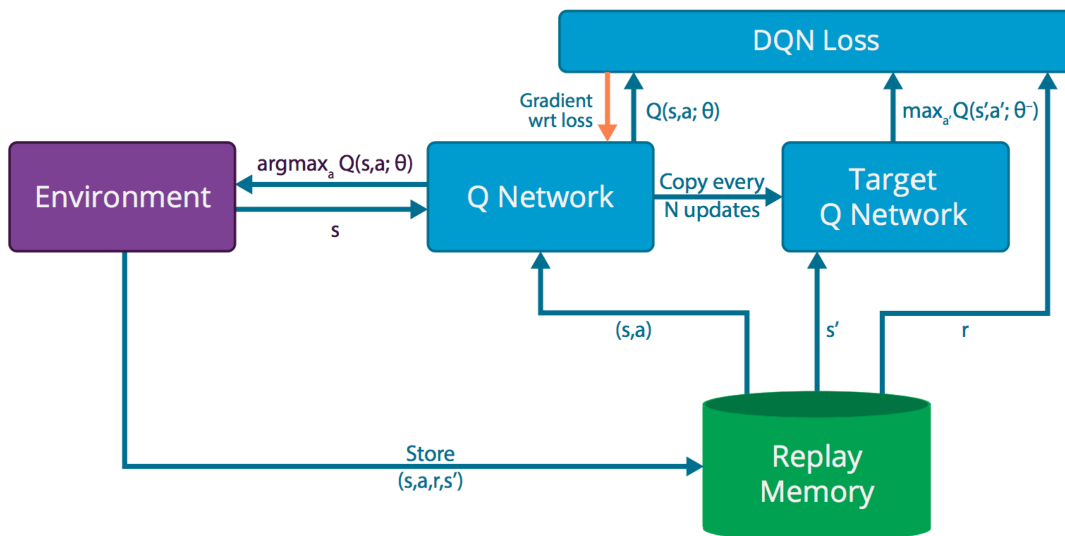


Figure 3.4: Experience replay implemented in DQN algorithm. Reproduced from Nair et al. (2015).

Improvements to ER have been made since, with a special emphasis on Prioritized Experience Replay (PER), currently used as the state of the art methodology in DRL algorithms, which we will explore next.

## 3.4 Prioritized experience replay

In previous ER methods, experiences were either sampled backwards, such as in the original proposal by Lin (1991), or more commonly sampled at random, therefore assuring the premise of independent identically distributed samples required to guarantee convergence of the gradient descent algorithm (Mnih et al., 2015). However, empirical experimentation shows that attributing an importance score to each experience and using it to steer the

sampling process by focusing on specific experiences leads to greater data efficiency in DRL algorithms (Schaul et al., 2015).

The notion of prioritizing experiences is already seen in prioritized sweeping (Moore and Atkeson, 1993). Prioritized sweeping aims to increase data efficiency in model-based algorithms by prioritizing dynamic programming sweeps to guide the exploration of the state space. It focus only on sweeps which are “interesting”, meaning it produces a large change in the state absorption probability. If the change produced is small, it attributes less urgency to update its predecessors state, and if its high it attributes a high urgency (Moore and Atkeson, 1993). Similarly, in PER, we attribute a high priority for the experiences which can cause a greater impact on the updates of the value function.

Upon entering the buffer, an experience is assigned a priority equal to the maximum existing priority plus a small constant $\epsilon$. This ensures every experience is visited at least once. When it is first sampled, the experience priority is updated to the magnitude of the TD-error, seen in Equation 2.6, which stands as a proxy for surprise. Greedly selecting experiences based on priority would lead to loss of diversity and overfitting to a smaller group of experiences. To mitigate this issue, the probability of sampling a transition $i$ is defined as a softmax distribution

$$Pr\{I = i\} = \frac{\rho_i^\alpha}{\sum_k \rho_k^\alpha} \tag{3.6}$$

where $\rho_i > 0$ is the priority of transition $i$. The exponent $\alpha$ controls for how much priority is used, with the uniform case being $\alpha = 0$ (Schaul et al., 2015).

PER have been successfully applied in the ALE benchmark environments, and is currently included in state-of-the-art algorithms that make use of ER.

After reviewing DQN and its main variations, we will move on to discuss how these algorithms apply to the the context of multiagent learning, our main topic of research.

# Chapter 4

# Multiagent Settings

In this chapter, we introduce the multiagent settings for RL problems. We introduce the more commonly used model, Markov Games, discuss the major research areas of stability, goal formulation and knowledge sharing, and review some of its applications.

## 4.1 Markov games

RL can be applied to single agent or multiagent scenarios. The intersection of multiagent and RL is called MARL (Shoham et al., 2007). MARL has strong connections with single-agent RL, game theory, evolutionary computation and optimization theory (Busoniu et al., 2008).

MARL is often formalized as a Markov game, a generalization of MDP to multiple agents (see Equations 2.1 and 2.2. It is represented by a set of players $P$, a set of states $S$, a set of joint actions $A$, where $A = A_1 \cdot A_2 ... \cdot A_n$, with $A_i$ being the finite set of actions available to each agent. The transition function

$$p(s' \mid s, \bar{a}) = Pr\{s_{t+1} = s' \mid s_t = s, \bar{a}_t = \bar{a}\} \tag{4.1}$$

and reward function

$$r(s, \bar{a}) = \mathbb{E}[r_{t+1} \mid s_t = s, \bar{a}_t = \bar{a}]. \tag{4.2}$$

are similar to the regular MDP, but the action $a$ is replaced by the joint action $\bar{a}$ . The joint action is a combination of the selected actions of all agents at each step:

$$\bar{a} = a_1, a_2, ..., a_n. \tag{4.3}$$

The goal of an agent is to learn the optimal policy. Policies can be deterministic, mapping state to actions as in $\pi : A \rightarrow S$, or stochastic, outputting the probability of

selecting an action given a state as $\pi : A \times S \to [0, 1]$. A Markov game is represented in Figure 4.1.

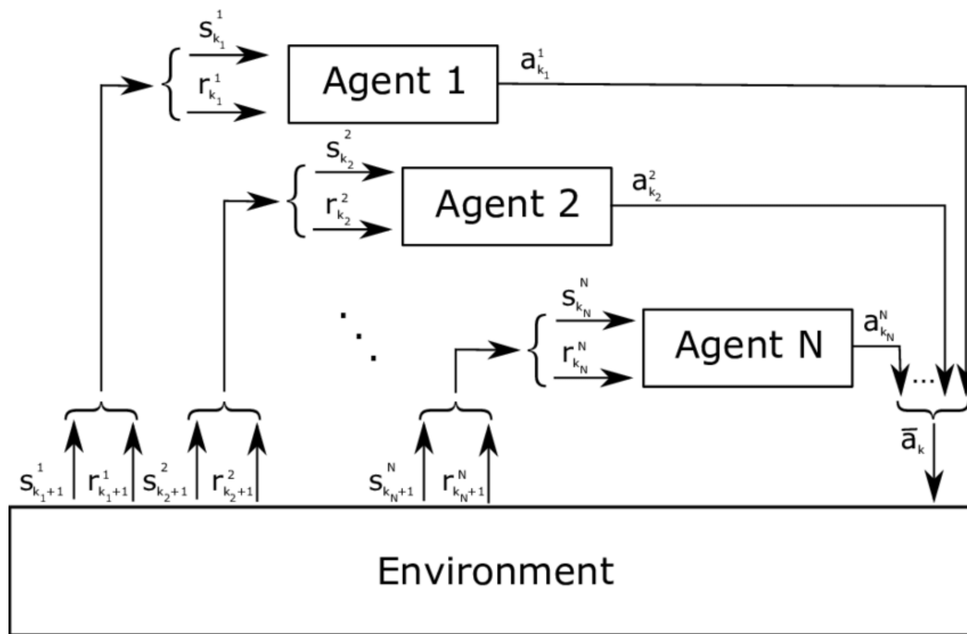


Figure 4.1: Representation of a Markov Game. The environment responds to a joint action. The new state and reward obtained is particular to each agent. Reproduced from Chincoli and Liotta (2018).

The settings can be classified as fully cooperative, fully competitive or somewhere in between, which comprises a wide spectrum of scenarios. Fully competitive two agents scenario is equivalent to a zero-sum game, where $r_1 = -r_2$ for two agents, whereas fully cooperative games have identical rewards $r_1 = r_2$.

More commonly, the rewards structure lies between the fully cooperative and fully competitive extremes. Variations of Markov games can include a single global reward function, or a combination of global reward function and local reward functions. Local rewards can be fully correlated, as in the extreme cases described above, partially correlated or uncorrelated.

From each agent's perspective, the environment is non-stationary due to ongoing learning from other agents. It creates a moving target problem, which breaks the Markov property - for an agent, its best policy may change as other agent's policies changes. This constitutes the biggest problem in MARL and prevents single agent RL convergence properties and guarantees to be extended to the multiagent scenario, despite its common use in practical applications.

One solution to the the dynamic nature of the MARL environment is to model the behaviors of other agents, predicting their choice of action at each state. This ability is a characteristic trait of human species, believed to be rooted in a specific region of the

brain called mirror neurons, and considered to be crucial to how we establish relationships and form cooperation and competition (Kilner and Lemon, 2013). Artificial autonomous agents modelling other agents is currently a prominent research field, with no consensus regarding the open problems (Albrecht and Stone, 2018).

The existing algorithms for MARL face a trade-off between stabilizing learning dynamics and prioritizing convergence, or continuously adapting to the dynamic nature of the environment. The decision of which type of MARL algorithm to apply mainly depends on the frequency and nature of the agents interactions, the correlation between their reward functions and the targeted learning goals. MARL algorithms should not be totally independent of other agents, nor just track their behavior without concerns for convergence (Busoniu et al., 2008).

Experience-buffer based RL methods, as DQN and Deep Deterministic Policy Gradient (DDPG), are incompatible with non-stationary environments. Stored transitions can quickly become outdated and no longer representative of the current dynamics. Proposed solutions includes mapping state-action pairs to decaying temperature values (Palmer et al., 2018), using a fingerprint to tag the age of transition sampled or using a variant of importance sampling to correct obsolete data (Foerster et al., 2017).

In real world applications, the scenario of sparse interactions is more commonly found. This has induced the proposal of dual nature learning agents, which can learn a policy at both the individual level, through single agent learning algorithms, and at a group level, and decide when to use each policy based on the expected interaction at each step (De Hauwere, 2011; Hu et al., 2015). Dual nature agents can benefit from state-of-the-art single agent algorithms and the vast multiagent research that leverages game theory and third-person reasoning to model the relationships between agents.

## 4.2 Learning goals

A formal statement of the multiagent learning goal is a central problem in MARL. While in single agent scenario the goal is to maximize total reward obtained over time, this is not always a possible or desired goal in MARL.

The traditional goal in Markov games is to achieve an equilibrium, or more commonly a Nash equilibrium (Hu and Wellman, 2003). Consider $U$ the utility, or the total reward an agent can obtain by following a policy, and $\pi = (\pi_1, ..., \pi_n)$ a policy profile that groups the policy of all agents. Nash equilibrium can be formally defined as

$$\forall_i \forall \pi'_i : U_i(\pi'_i, \pi_{-i}) \leq U_i(\pi), \tag{4.4}$$

where no agent can improve utility by unilaterally changing its policy.

Nash equilibrium describes a status quo. Equilibrium was quickly adopted as the goal for many initially proposed MARL algorithms, but it has many limitations, such as:

- Non-uniqueness: multiple Nash equilibrium can exist;
- Incompleteness: does not specify behaviour for off-equilibrium paths;
- Sub-optimality: not the same as utility maximization;
- Rationality: assumes all agents are perfect utility maximizers.

An alternative for equilibrium is Pareto Optimum, defined when there is no other profile $\pi'$ such that an agent cannot improve its policy without making another agent's policy worse off:

$$\forall_i : U_i(\pi') \geq U_i(\pi) \text{ and } \exists_i : U_i(\pi') > U_i(\pi). \tag{4.5}$$

Three other common goals are social welfare, fairness, and welfare/fairness:

$$\text{Welfare}(\pi) = \sum_i U_i(\pi), \tag{4.6}$$

$$\text{Fairness}(\pi) = \prod_i U_i(\pi), \tag{4.7}$$

$$\text{Welfare/Fairness}(\pi) = \frac{\sum_i U_i(\pi)}{\prod_i U_i(\pi)}. \tag{4.8}$$

The application of MARL agents to problems that involve multiple cooperative agents, such as in sociological contexts, is leading the research on goals that are associated with a more equal distribution amongst agents. Welfare over fairness is equivalent to the harmonic sum of all utilities and it is a viable option to seek both maximization and equality. Goals can also be customized for specific safety concerns, such as constraining utility to lower or upper bounds, or targeted optimality (Albrecht and Ramamoorthy, 2012).

## 4.3 Applications

Cooperative multiagent applications can benefit from the speed of parallel computation, exploiting the decentralized structure of a task, and accelerated learning by knowledge sharing amongst agents, teaching or imitation. Multiagents can also add robustness to a model by having one agent taking over another when it fails (Busoniu et al., 2008).

While the application are many, we emphasize distributed control, resource management, collaborative decision support systems, negotiation, robotic teams and multiplayer

games (Busoniu et al., 2008). Smart grid, wireless networks and internet of things are possible applications that can be significant to economy in the near future.

Multiplayer games are still the main test bed for MARL agents. It has seen a recent surge with the deployment of agents able to successfully play online multiplayer games of the e-sport category, such as DOTA 2 (Fernandez and Mahlmann, 2018), where two teams of five players compete in a virtual arena, or Starcraft 2 (Vinyals et al., 2017), a role playing strategy game that can include one or multiple teams of players competing against each other. The popular RoboCup, represented in Figure 4.2, has driven research into robotics and MARL for more than two decades.

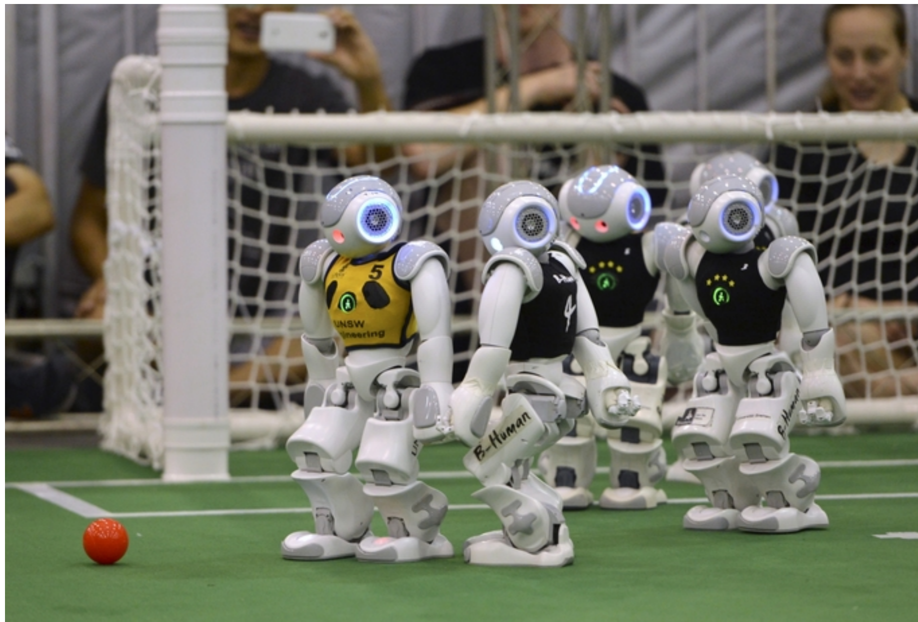


Figure 4.2: Soccer standard platform league match in RoboCup (2015).

Other settings involving multiplayer games have been proposed as a scenario for which to compare and benchmark solutions to the MARL problem. Two of these are worth noting. The first is Pommerman, a multiagent environment based on console game Bomberman, which contains both cooperative and competitive traits (Resnick et al., 2018), and recently ran a competition in NeurIPS 2018 (formerly known as NIPS). Second is Marlo, a MARL environment based on Project Malmo and the Minecraft engine, and sponsored by Microsoft (Johnson et al., 2016). It also held a competition online, kickstarted in an AAAI 2018 workshop.

These competitions are designed to motivate the creation of agents that can successfully work as a team, which includes amongst others the ability to devise joint plans, model teammates and opponents, reason in game theory and communicate effectively.

While applying MARL algorithms to games serves as a benchmark that allow comparisons between evolving methods, the biggest challenge lies in extending those applications

to real-life tasks. In those, scalability and robustness to imperfect observations are required, and both characteristics have been shown to be problematic to achieve in MARL algorithms.

Another challenge we've discussed is defining MARL goals that involve not only reward maximization, but also fairness, safety and other auxiliary objectives. Coordination between autonomous vehicles that learn through RL is an example of applied MARL which has gained traction in the last years, and has functional safety as its main goal (Shalev-Shwartz et al., 2016).

## 4.4 Knowledge sharing

A key benefit in cooperative multiagents setting is to benefit from other agent's knowledge by sharing experiences, an approach which has been extensively studied in the MARL literature.

Lin (1992) points out that the success of learning in a trial-and-error process relies on the agent's luck in first achieving the goal by chance, which is correlated with how delayed is the reward. This learning barrier is one of the main issues of slow learning time, and could be overcome by learning expertise directly from external experts. Teaching could direct the learner either to explore a promising part of the search space which contains the goal states, important when the search space is large and thorough search is infeasible, or help him avoid getting stuck in local maxima.

In Whitehead (1991), the author observes that in nature, intelligent agents exist in a cooperative social environment, that structures and guides learning. In this context, learning involves as much information transfer as it involves discovery by trial-and-error. So logically methods to share knowledge should be pursued with equally measure to methods that learn through trial and error. He points that even if individuals are not able to make discoveries at a useful rate, the inherent parallelism in large populations can overcome the complexity of the search, and the group could accumulate knowledge and adapt at an acceptable rate as opposed to lone individuals.

Whitehead (1991) proceeds with a complexity analysis of the cooperative mechanism of ES, establishing tentative upper bounds, but gives no empirical evaluation of the method proposed. The methods are further investigated in Tan (1993). Tan poses the questions: "Given the same number of reinforcement learning agents, will cooperative agents outperform independent agents who do not communicate during learning?" and "What is the price for such cooperation?". He sets to answer those questions and puts forward a validated theoretical framework for ES. A main concern raised in this work is the cost of

communication. The benefits gained from sharing should outperform the cost added due to the increased communication between the agents.

Tan (1993) proposes three main approaches for sharing knowledge: (i) sharing a learned policy, between a more knowledgeable agent and a novice one; (ii) sharing entire episodes, sequences of experiences; and (iii) sharing experiences. His main thesis is that if cooperation is done intelligently, each agent can benefit from other agents' information.

A similar classification is also put forth by Boutsioukis et al. (2011), which proposes different forms of transferring knowledge, including:

- value functions;
- policies;
- rules;
- action subsets;
- shaping rewards;
- experiences as tuples $(s_t, a_t, r_t, s_{t+1})$.

The simplest way of cooperating by sharing a learned policy is having one single policy for all agents. This centralized control approach, while a simple and effective solution, has the highest burden in terms of communication, as all agents need to communicate back and forth with a centralized controller at every step taken. On the other spectrum, each agent can keep its independent policy, and only consider assimilating another agents' policy when it does not have confidence in certain actions in its own policy. This approach of sharing a part of the policy, in form of an action advice, is successfully explored in the teacher-student framework put forward by Torrey and Taylor (2013).

In the context of sharing episodes, a Monte-Carlo agent that learns from episodes of two or more cooperative agents could significantly speed up learning by multiplying the samples it learns from. However, as discussed in Lin (1992), learning from episodes generated from a different behavior policy invalidate the convergence theorem and might lead to worst performance (Sutton and Barto, 2018). Importance-sampling can be used to adapt the behavior policy to the learned policy (Doucet et al., 2001). But the longer the sequence of actions, the higher the probability that the paths diverge, making the episode shared be considered useless for learning.

The third approach proposed relates to ES. The shared experiences can either be used to perform a single learning step of the value function, or be stored in a local memory to be reused several times. In the ES approach, atomic transitions are shared, instead of entire episodes containing sequential transitions. Since in an atomic transition the policy sampling is limited to one action, it is more likely the experience can be useful for learning without affecting the convergence of the algorithm, as it occurs in the episode sharing approach.

ES can be easily extended to model-free learning algorithms with ER. The shared experiences are added to the learning agent's buffer, and can be periodically used to update the value function. This is foundation of our proposal, which we will discuss in Chapter 5.

# Chapter 5

# Proposal

The previous chapter introduced methods of knowledge sharing between cooperative agents in multiagent domains. Among these methods, we have seen ES, which is easily extendable to DRL algorithms that make use of experience replay techniques and can be implemented in scenarios where agents interact in sparser intervals.

In this chapter, we present MACES, a cooperative multiagent model that makes use of ES to accelerate learning. Within MACES architecture, we propose four different methods of ES: (i) Naive ES; (ii) Focused ES; (iii) Prioritized ES; (iv) and Prioritized Focused ES.

The environment in MACES has the following characteristics:

- **Multiagent**: two or more agents learn concurrently, and are allowed to cooperate to achieve their goal;
- **Stationary**: from the perspective of the agent, the transition and reward functions do not undergo changes over time;
- **Stochastic**: transition and rewards may be determined by a probability function distribution;
- **Continuous state space**: state space is represented by a set of variables that may be continuous. For images, the $[0, 255]$ discrete interval that represents the color intensity of a pixel is converted to a continuous $[0, 1]$ interval.

The agents have independent MDPs with similar goals. Experience in a MDP is defined as a tuple $(s_t, a_t, r_t, s_{t+1})$, representing a transition taken by an agent from one state to another and the response received from the environment. The characteristics of the agents implemented in MACES are as follows:

- **Autonomous**: agents are independent learners, with individual goals;
- **Homogeneous**: all agents share the same state space and action space;

- **Discrete action space**: agents have a discrete set of actions available.

As independent learners, each agent is formalized by its own MDP, with individual actions and transitions, as opposed to the common multiagent formulation of Markov Game with joint actions. This gives us the advantage of ignoring the interactions between the agents, allowing us to compare the results of cooperation between two or more learning agents against a single learning agent. This relaxation of the multiagent setting has been commonly used in the study of transfer learning in MARL algorithms (Busoniu et al., 2008; Price and Boutilier, 2003).

The MACES architecture is represented in Figure 5.1. It shows the relationship between two concurrently learning agents in replicated environments. The dotted arrows represent the experiences flow. The agents exchange information through a Requests Board ($RB$), a common data structure accessible by all agents, similar to the blackboard architecture described by Wooldridge (2009). Each agent has a replay buffer, which stores the experiences, and an inbox, a temporary repository to store experience received from other agents until they can be permanently incorporated into the replay buffer.

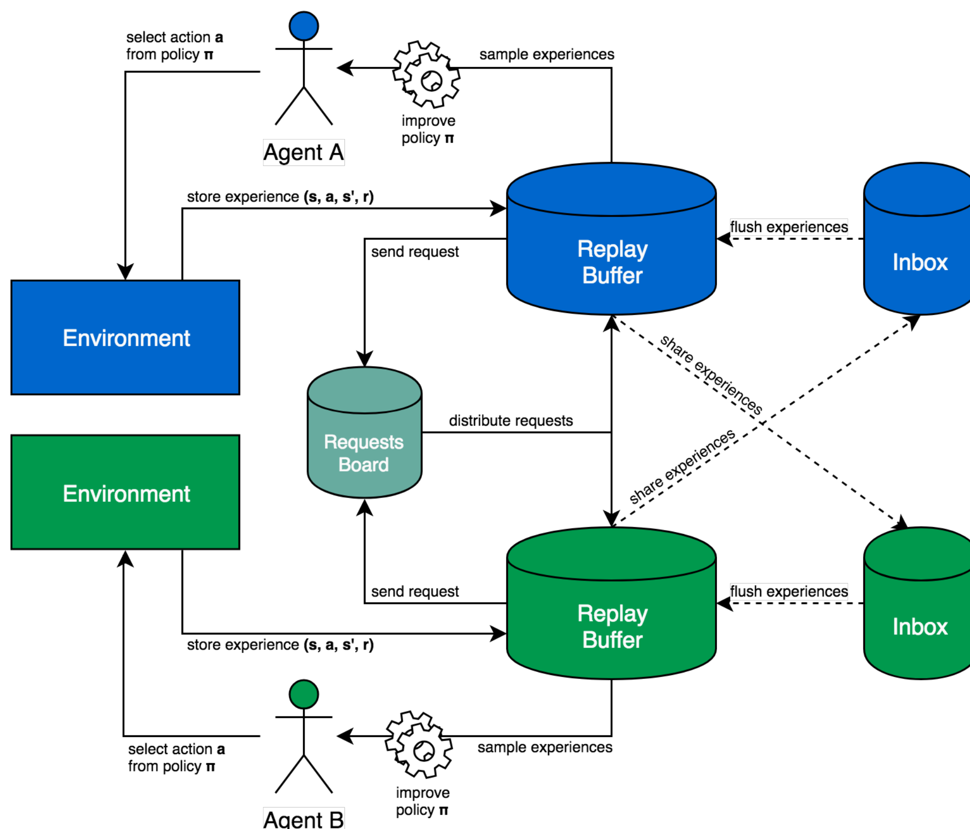


Figure 5.1: Experience sharing architecture showing cooperation between two agents. The environment, replay buffer and inbox (shown in color) are unique for each agent, and the requests board (shown in gray) is shared among them. It can be extended to any number of agents.

All methods in MACES can potentially be applied to any model-free DRL algorithm which makes use of experience replay, including the popular DQN (Mnih et al., 2015) and DDPG (Lillicrap et al., 2015). However, at this work we only present the implementation with DQN.

## 5.1 Experience sharing methods

In MACES, the agents learn simultaneously, and at each round of ES, can play the role of teacher, learner or both. For simplification, we explain the proposal with two agents, but it can be extended to any number of agents. When one agent is assigned as teacher, all others are assigned the role of learner, and therefore are entitled to receive experiences from the teacher agent.

The ES process can be divided in two stages. In the episode stage, each agent incorporates experiences received from the last sharing stage into its buffer, and executes the episode. After completing the episode, the agent issues a new request for help to a public requests board. The stage ends when all agents have completed their episodes. In the sharing stage, all agents alternatively assume the role of teacher. As a teacher, the agent verifies if there are available requests in the requests board which are not its own. If there are, it fulfills the request by sending a batch of experiences to the requesting agent's inbox.

The batch size is limited to the minimum between $\kappa$ and the replay buffer size. The variable $\kappa$ represents the maximum size of an ES batch, and is a hyperparameter of the model. In real-world applications, the batch of experiences shared is bounded by the communication bandwidth available between two agents.

ES is typically performed by frequently sharing small batches of experiences. By constrast, we will focus on the episode by episode approach, allowing for a more varied range of experiences to be included in the batch before the sharing occurs and limiting the communication between the agents to once each episode. The pseudocode is described in Algorithm 3.

We propose four methods of ES, and proceed to validate them empirically. The methods differ mainly in how the request is composed by the requesting agent, and how experiences to be shared are selected by the teacher agent. The methods are detailed in Sections 5.1.1, 5.1.2, 5.1.3, and 5.1.4.

### 5.1.1 Naive ES

Requests contain no details. Experiences shared are randomly sampled from the teacher's replay buffer.

**Algorithm 3** Experience sharing

```
1: initialize environment and agents
2: initialize empty requests board RB
3: while not (all agents completed task) do
4:     parallel for agent 𝒜 in the environment do
5:         𝒜 add experiences from inbox to buffer
6:         𝒜 plays episode
7:         𝒜 adds new request ℛ to RB
8:     end parallel for
9:     parallel for agent 𝒜 in the environment do
10:        check RB for available requests from other agents
11:        for request ℛ in available requests do
12:            𝒜 sample batch of experiences ℬ matching ℛ
13:            𝒜 places ℬ in requesting agent's inbox
14:        end for
15:    end parallel for
16:    clear RB
17: end while
```

### 5.1.2 Focused ES

In Focused ES, when forming the request, the agent swipes its buffer to identify regions of the state space that have been poorly explored. This can be achieved by maintaining a second structure parallel to the buffer called an occupancy grid.

The occupancy grid is a three dimensional $n \times m \times o$ tensor, where $n$ is the number of actions available to the agent, $m$ is the number of variables in the state, and $o$ the number of bins in which the variables are discretized. This data structure is used to count the number of times each region of the state space has been visited.

Whenever a new experience is added to the buffer, the agent discretizes the state using state aggregation, which consists of binning each continuous variable and combining the resulting bins. Using 10 bins per variable, a state represented by a vector of size 4 has $10^4$ possible categories, given by the combination of the bin positions for each variable.

The experience is allocated to the occupancy grid according to the discretized state and the action. This process can also be extended to continuous action spaces by previously combining the state and action variables, and using a matrix $n \times o$ as the occupancy grid where $n$ corresponds to the sum of the variables that represent state and action.

The occupancy grid allows for fast recovery of data and fact-checking to verify the grid occupancy. The impact of miscategorizing an experience in Focused ES is low, so we favor regular state aggregation over more complex approaches to perform discretization. However, we also evaluate a variant of Focused ES using tile coding with different number of tiles, and compare it to the regular state aggregation method.

In more complex multidimensional state spaces, a block reduce averaging procedure is used prior to discretization to downsample the tensor input into a vector. This procedure is also known as average pooling and is widely used in DNNs. It works by sliding a filter over the original tensor. The values selected by the filter are averaged and projected into a lower dimensional output, as shown in Figure 5.2. By applying a $4 \times 28 \times 28$ filter to a $4 \times 84 \times 84$ tensor, we can reduce it to a $1 \times 1 \times 3$ tensor, which can be further flattened to a vector of length 9. Using state aggregation with 4 bins per variable results in $4^9$ possible ways to classify each state perceived by the agent.

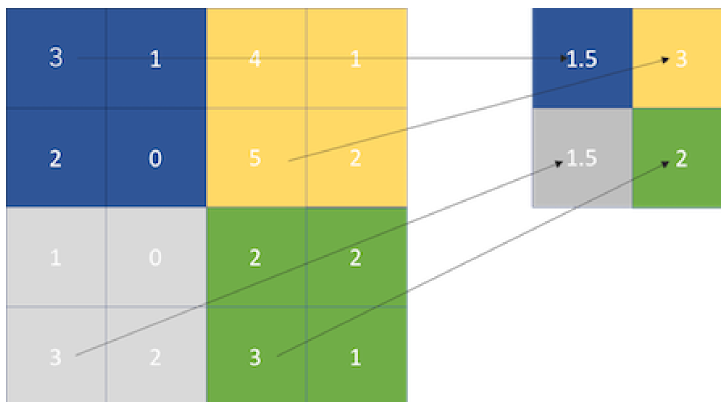


Figure 5.2: Average pooling performed on a 4x4 input with a 2x2 kernel. The output result is the 2x2 matrix shown in the right. Reproduced from ReNom (2014).

Storing an experience corresponds to step one in the schematic process shown in Figure 5.3. In step two, the agent selects a mask of the occupancy grid where each position is marked as unexplored if the number of experiences in the position is less or equal a threshold $\zeta$. By varying $\zeta$ we can control for how many experiences defines what it means for a region to be unexplored.

In step three, the teacher agents who receives the request use the request mask to identify experiences in its buffer that belongs to the unexplored regions of the student's state space. This procedure requires both agents to have the same state and action space, being suitable only to homogeneous agents in similar environments. The experiences selected in step three are randomly sampled to form the batch of experiences to be sent to the inbox of the requesting agent, which are later added to its buffer (step four).

### 5.1.3 Prioritized ES

As in Naive ES, requests contain no details. Experiences to share are sampled using priorities to define the probability of an experience being sampled. The priorities used for experience sharing are the same defined in the Prioritized Replay method introduced by Schaul et al. (2015). The priority of an experience is defined as the TD-error calculated

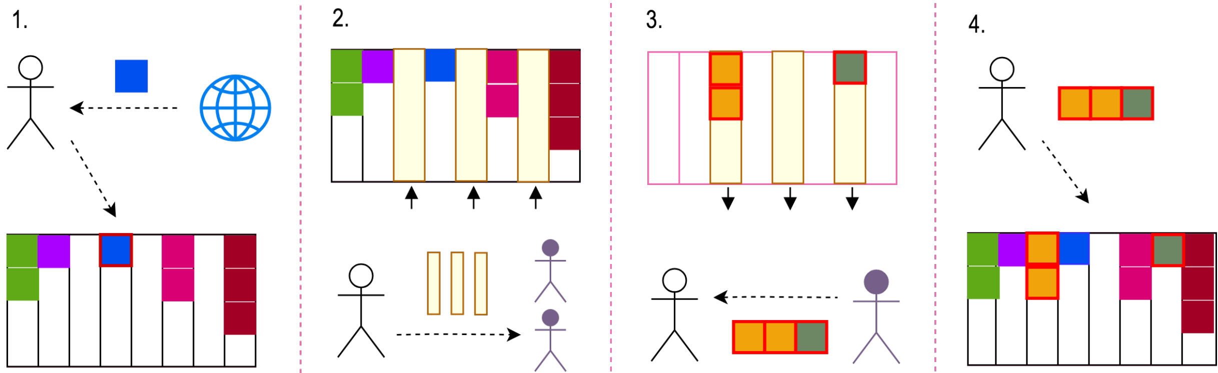


Figure 5.3: Schematics of the Focused ES implementation. The large rectangles exemplify the occupancy grids. Each experience, represented by a colored square, is allocated to its respective place in the occupancy grid according to the discretized state. The highlighted bins in stage 2 represents the mask identifying the unexplored regions in the student's occupancy grid.

when the experience is used for learning. The TD-error is the difference between the total return expected to be obtained from the experience and the actual return obtained (see Equation 2.6). It can also be understood as a measure of surprise, or how unexpected the experience is to the agent that lives it. Since the teacher agent has no access to the student's other than the request, it calculate priorities based on its own action-value function.

### 5.1.4 Prioritized Focused ES

The request process is similar to the Focused ES, described in Section 5.1.2. The teacher agent proceeds to select the experiences based on the request, using the occupancy grid. In the last stage, instead of randomly sampling to form the batch of experiences to be sent, the priorities assigned to each experience according to the PER method are used instead to determine which experiences will be selected for sharing.

## 5.2 Baseline algorithm

The underlying RL algorithm used in MACES is DQN (see Algorithm 2). Several improvements to DQN have been introduced over the last few years, and the most important of them were considered in the baseline implementation, approaching the state-of-the-art technique.

To test MACES, we conducted experiments with six different versions of the algorithm. Two are single agent implementations, to be used as baseline, and four are multiagent implementations enhanced by the proposed ES methods.

The most relevant modification is using the target network to accrue the value of the next state and action when bootstrapping, introduced in Van Hasselt et al. (2016). This modification to the original DQN is introduced as a new algorithm, Double-DQN, which we will call here DQN for simplification. We also applied soft updates to the target network, using a parameter $\tau$ which controls how much of the learning network is merged with the target network at every step (Lillicrap et al., 2015).

A batch of experiences is randomly sampled from the replay buffer at every step, and used to calculate the loss and update the weights of the network accordingly. Exploration is done using $\epsilon$-greedy policies, with epsilon reduced at every step by a linear rate. The linear rate is calculated by setting the final epsilon value, a minimum rate of exploration, and a number of frames to decay. The epsilon decay rate is given by the number of frames divided by the initial epsilon minus the final epsilon.

To approximate the action-value function we implement a multilayer perceptron, with one input layer, two hidden layers and an output layer. As in DQN, the neural network approximates the action-value function $Q$, mapping a state to action values. The input layer has four neurons, equivalent to the state size, and the output layer has two neurons, equivalent to the number of actions. There are two hidden layers of 16 and 8 neurons respectively, which uses rectified linear units as the non-linear activation function. This architecture is a modification of the original DQN publication (Mnih et al., 2015), with significantly less degrees of freedom due to the simplicity of the task.

For problems where the state space is represented by an image, a convolutional neural network is used instead, shown Figure 5.4. The layers are further detailed in Table 5.1.

Two versions of the baseline are used. The first is the DQN, as described above. The second, which we call Deep Q-Network with Priority Replay (DQN-PR), uses prioritized replay to decide which samples to replay at every learning step. When using ER, each sample is assigned a maximum priority when entering the batch, ensuring it is sampled at least once. Every time an experience is sampled, its priority is updated according to the TD-error calculated for it. The TD-error represents how much of an impact an experience had in the weight adjustment done in a particular step. It is also a proxy for how surprised the agent is in experiencing that transition. Its implementation is inspired on neuroscience studies reporting similar behavior in rodents (Schaul et al., 2015).

The neural network implements the Adam optimizer, and its parameters, $\beta_1$ and $\beta_2$, are set to the default values $\beta_1 = 0.9$ and $\beta_2 = 0.999$ considered optimal for the majority of problems regarding neural networks (Kingma and Ba, 2014). Clipping the gradients

Table 5.1: Convolutional neural network detailed layers.

| Input | Neurons | Kernel | Stride | Output | Layer Type |
|---|---|---|---|---|---|
| $84 \times 84 \times 4$ | 32 | 8 | 4 | $20 \times 20 \times 32$ | Convolutional |
| $20 \times 20 \times 32$ | 32 | 8 | 4 | $9 \times 9 \times 64$ | Convolutional |
| $9 \times 9 \times 64$ | 64 | 8 | 4 | $7 \times 7 \times 64$ | Convolutional |
| $7 \times 7 \times 64$ | 512 | - | - | $1 \times 512$ | Global Average Pooling |
| $1 \times 512$ | 4 | - | - | $1 \times 4$ | Fully Connected |

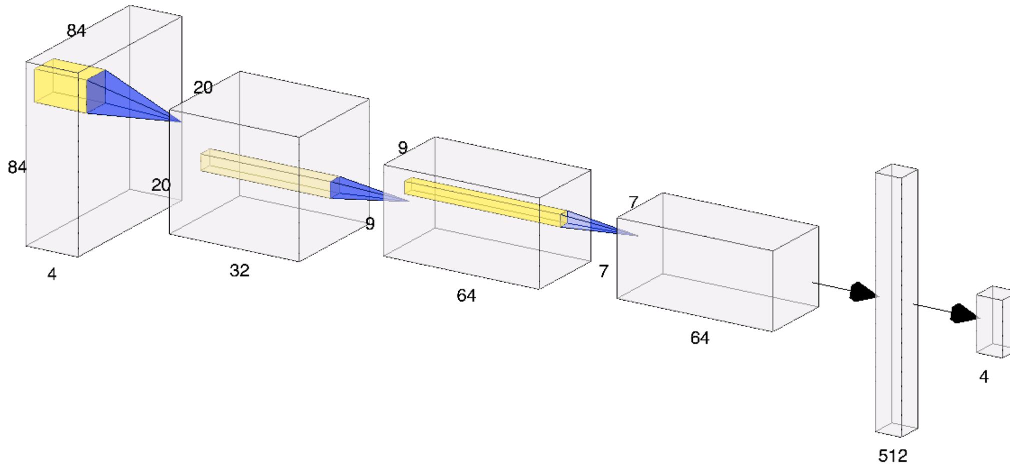


Figure 5.4: Convolutional neural network architecture.

for the neural network was also attempted, but it led to inferior performance, and was not considered in the implemented baseline. The remaining hyperparameters were optimized by coarse grid search. The complete list of hyperparameters selected for the baseline algorithm are given in Appendices A and B.

## 5.3 Implementation

To support the research and experiments, a Python library named fasterRL has been created. We will briefly introduce it to reader. The library is coded in Python 3.6, and supports future versions with backwards compatibility.

Its main focus is on experiment replication. All the required hyperparameters are defined in a json file, which is saved to allow future replication. The details for the experiment are recorded in several levels. Learning variables such as q-values, epsilon, and rewards per step, are recorded as tensorboard files, and mainly used for a detailed analysis of the learning dynamics and debugging. Experiments final results such as number

of episodes to complete the task, average number of steps and wall clock speed are averaged over many trials and recorded as json files. These are the main metrics that represent a RL algorithm performance and can be used to compare different algorithms. The use of fasterRL is illustrated in Listing 5.1.

```
from fasterRL.common import experiment

params = {
    "LOG_LEVEL": 2,
    "PLATFORM": "openai",
    "ENV_NAME": "FrozenLake-v0",
    "METHOD": "QLearning",
    "NUM_TRIALS": 3,
    "LEARNING_RATE": 0.3,
    "GAMMA": 0.99
}

exp = experiment.UntilWinExperiment(params)
exp.run()
```

Listing 5.1: Sample code to run experiment. Parameters can be passed as a separate json file or in code as python dictionaries.

PyTorch 4.0 is used as the automatic differentiation framework to support neural networks implementation. The dependency is encapsulated, to allow the library to be extended to other frameworks such as TensorFlow and Caffee. Multiagent implementation is provided in sequential and parallel approaches.

The implemented algorithms that are commonly presented in the literature are:

- Q-Learning
- Sarsa
- MonteCarlo
- Policy Gradients
- Cross Entropy
- Reinforce
- Deep Q-Networks
- Double Deep Q-Networks
- Deep Deterministic Policy Gradient
- Actor-Critic

- Advantage Actor-Critic

Of the algorithms listed, Deep Q-Networks and its variant Double Deep Q-Networks were used in the experiments. Customization options are available for state-of-the-art methods, including:

- Discretization with aggregation (for state and/or action space)
- Discretization with tile coding (for state and/or action space)
- N-steps for off-policy methods
- Importance Sampling
- Gradient Clipping
- Priority Replay
- Multiagents
- Experience Sharing

FasterRL is designed to be platform-agnostic, allowing algorithms to be tested against a wide range of environments in different open-source RL platforms. It currently supports the RL environment platforms OpenAI, GymMinecraft and Marlo. Planned roadmap includes support for PyBullet-Gym, RoboSchool, VizDoom, DeepMindLab and Industrial Benchmark. The code is available at `https://github.com/lucasosouza/fasterRL`.

ES introduces an experience sharing model for cooperative multiagent RL settings. In Chapter 6, an empirical validation of MACES and the methods proposed is conducted.

# Chapter 6

# Empirical Validation

The proposed model was evaluated empirically in two different simulated environments, featuring a classical control problem and a navigation problem. We first discuss the experimental procedures used, and proceed to show the results achieved and discuss its limitations.

## 6.1 Experimental procedures

Directly comparing two algorithms in a single trial is not reliable due to the non-deterministic nature of both the agent's function and the environment function. The agent's action selection, experience buffer sampling and neural network initialization are all in part stochastic processes. The environment's transition function is likewise given by a probability distribution. Therefore, in order to compare two or more algorithms, the experiment is repeated a number of times with different random seeds, and the distribution of the results are compared, as proposed in Henderson et al. (2018b).

Each repetition is called a trial. A trial ends when all agents completes the task. The main performance metric used for evaluation is the ETC. In the single agent variant (baseline), a trial adds only one sample to the distribution. In the multiagent variant, the results of all agents are added to the distribution. The number of trials executed were 100 for single agent and 50 for multiagent variant with two agents; as a consequence, the distribution for each variant tested is composed of 100 samples.

Although 30 is typically deemed to be the minimum sample size required to apply large-sample statistics (Hogg and Tanis, 2009), considering the high variance of the sample results and seeking to provide robust outcomes, we've decided on using 100 as the sample size. The size of the distribution is enough to be considered representative of the entire population.

To compare the two distributions, we use the Kolmogorov–Smirnov (K-S) test, as suggested by Henderson et al. (2018b). K-S is a nonparametric goodness-of-fit test, used to determine whether two distributions differ. It is commonly used to compare two samples coming from two populations that might be different (Massey Jr, 1951).

The experiments were conducted in an Ubuntu powered Desktop, with an Intel i5-6500 3.20 GHz quad-core processor, 32gb RAM and NVIDIA GeForce 1070 graphics card.

## 6.2 OpenAI: Cart Pole

In this Section, we will review and discuss the experimental results of MACES applied to a RL classical control problem.

### 6.2.1 Environment description

The first environment evaluated is Cart Pole, a classic control problem introduced in Sutton and Barto (2018), using the OpenAI Gym library (Brockman et al., 2016).

Cart Pole environment, represented in Figure 6.1, consists of balancing a pole, attached by an un-actuated joint to a cart, that moves along a frictionless track. The goal of the agent is to apply force to the cart, so as to balance a pendulum standing on top of it. There are two discrete actions available, which corresponds to either applying a force of +1 to move the cart to the right or a force of -1 to move the cart to the left (OpenAI, 2018). There is also a version of this environment with continuous action space, which we will not cover in the experiments.

The episode starts with the pendulum upright, and it ends when the pendulum falls over to one of the sides. At every step, the agent receives a reward of 1 if the pendulum has not fallen to the side. There are maximum 200 steps available, so the maximum reward obtainable is 200. A task is completed when the agents achieves a stable optimal policy. In our experiments, that translates to obtain a reward of 199 or greater over 10 consecutive episodes. Evaluation is not done separately - the same trials used for training are used for evaluation, so the agent has to carefully consider the exploration-exploitation trade-off in order to achieve the goal.

The state perceived by the agent is defined by four continuous variables:

- the cart position, ranging from -2.4 to 2.4;
- the cart velocity, ranging from -Inf to Inf;
- the pole angle, ranging from - 41.8 to 41.8 degrees;
- the pole velocity at tip, ranging from -Inf to Inf.

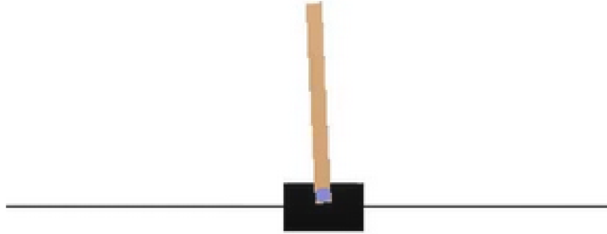

Figure 6.1: OpenAI CartPole environment

In our settings, no initial knowledge of the environment is allowed, including sampling random transitions from the environment to fill a replay buffer as seen in Mnih et al. (2015). As a consequence, the agent only starts to learn after its buffer reaches a number of experiences equal or greater than the learning batch size defined, which takes approximately between 1 to 5 episodes with the hyperparameters defined.

The performance of the agent is measured in terms of the number of episodes required to complete the goal, or ETC. The minimum ETC is 10, as the goal requires a moving average of the last 10 episodes. The maximum ETC allowed is 1000. If the agent is unable to reach the goal within the delimited number of episodes, ETC is set to 1000 and the trial is registered as a failure.

### 6.2.2 Results

With the experimentation procedure explained, we proceed to discuss the results achieved. We first compare multiagent variants DQN + Naive ES and DQN + Focused ES with single agent DQN baseline, and multiagent variants DQN-PR + Prioritized ES and DQN-PR + Prioritized Focused ES with single agent DQN-PR baseline. We aim to show that the cooperative multiagent variants can outperform the single agent baseline.

In Figure 6.2 we plot the samples from a single agent DQN versus a multiagent DQN with two agents sharing experiences. In all experiments, a normalized histogram and a kernel density estimation of both distributions are used to compare.

Results shows that multiagent DQN with Naive ES adds no improvement over the single agent DQN. However, multiagent DQN with Focused ES shows a significant improvement over the baseline. The Focused ES method has an average of 154.44 and a standard deviation of 111.92 ETC, compared to an average of 317.65 and a standard deviation of 176.64 ETC in single agent DQN, resulting in a 51.4% improvement in performance. A two sample K-S test rejects the null hypothesis that both samples are drawn

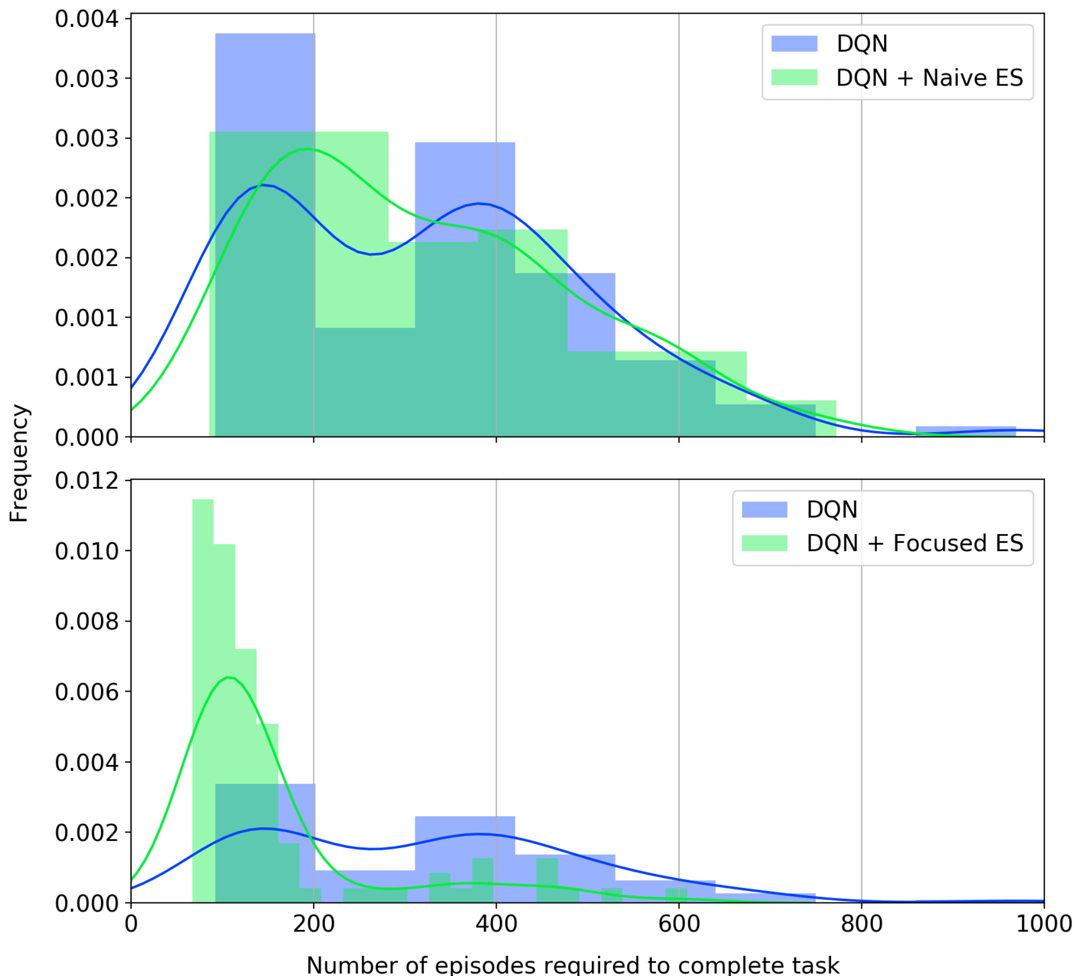


Figure 6.2: Comparison of single agent DQN with multiagent DQN with Naive ES and Focused ES.

from the same distribution, with a p-value of $3.70 \times 10^{-12}$. These results are shown in Table 6.1.

The same comparison is shown for the DQN-PR algorithms in Figure 6.3. Multiagent DQN-PR with Prioritized ES have a significant higher average ETC (459.63) compared to single agent DQN-PR (300.22), with 26 out of 100 samples failing to complete the task. We speculate that the regular stream of new experiences received being assigned maximum priority stops the agent from replaying old experiences, which are eventually discarded when the buffer reaches maximum capacity, before they are used for learning.

By combining the Focused ES method with Prioritized ES we can ensure only the most relevant experience are shared. DQN-PR with Prioritized Focused ES shows an improvement in performance of 31.1% over the baseline DQN-PR. As before, we apply a K-S test to test the hypothesis of both samples being drawn from the same distribution, which we reject with a p-value of $1.74 \times 10^{-4}$.

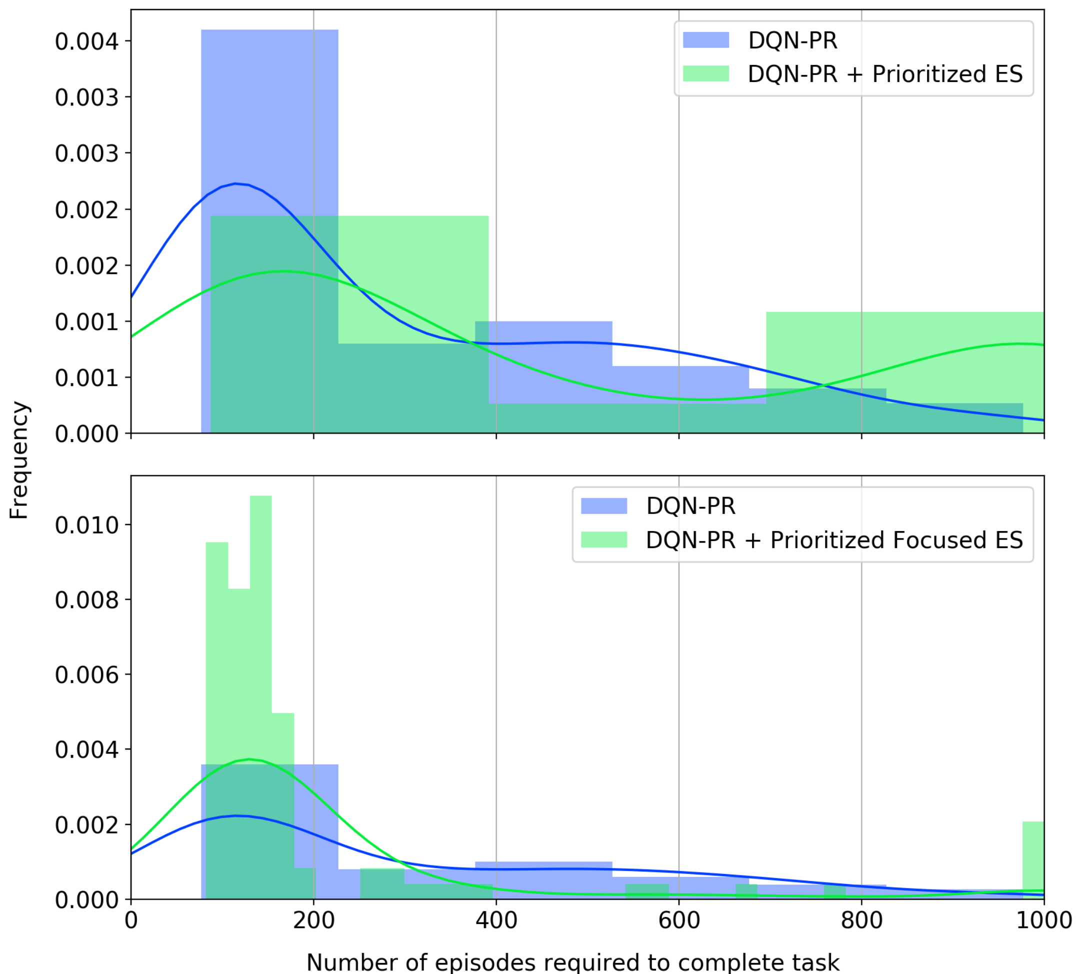


Figure 6.3: Comparison of single agent DQN-PR with multiagent DQN-PR with Prioritized ES and Prioritized Focused ES.

We directly compare the best approaches DQN + Focused ES with DQN-PR + Prioritized Focused ES in Figure 6.4. DQN + Focused ES distribution shows lower average (154.44) and lower variance, while DQN-PR + Prioritized Focused shows a long right-side tail distribution which pulls the average higher (206.87), with the agent failing to achieve the goal in 5 out of 100 samples.

We can better understand how focused experience sharing affects learning by analyzing the learning dynamics episode by episode. Figures 6.5 and 6.6 plot the average reward along episodes for the considered algorithms. In the plots, each line represents an average over 100 trials (with standard deviation shown as shades). To enhance presentation, all lines are shown up to 500 episodes. In the case of trials that completed the task before 500 episodes, we consider the last obtained reward to compute the average of subsequent episodes.

In Figure 6.5 we see how in DQN + Focused ES learning progress faster right from the

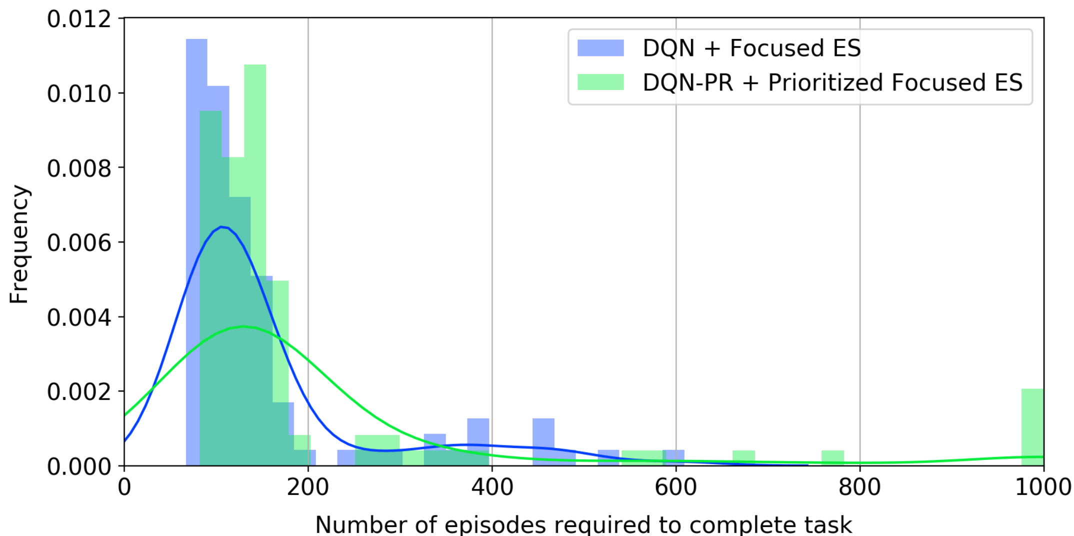


Figure 6.4: Comparison of multiagent variants DQN with Focused ES and DQN-PR with Prioritized Focused ES.

Table 6.1: Experiments results in Cart Pole environment.

| Method | ETC Mean | ETC Deviation | Trials Failed | ETC Improvement |
|---|---|---|---|---|
| DQN | 317.65 | 176.64 | 0 | - |
| DQN + Naive ES | 318.19 | 163.26 | 0 | -0.2% |
| DQN + Focused ES | 154.44 | 111.92 | 0 | +51.4% |
| DQN-PR | 300.22 | 246.05 | 0 | - |
| DQN-PR + Prioritized ES | 459.63 | 370.57 | 26 | -53.1% |
| DQN-PR + Prioritized Focused ES | 206.87 | 214.63 | 5 | +31.1% |

beginning, while in DQN + Naive ES progress is slower, even when compared to the the single-agent variant. The differences in performance are most notable after they reached a high level reward, around 175. Most of the variance in ETC in DQN and DQN + Naive ES can be explained by the time it takes to cover the last steps towards the target, leaping to the maximum reward of 200. DQN + Focused ES is able to overcome this last stage using fewer episodes.

A similar behavior occurs when priority replay is added, seen in Figure 6.6. In this case, DQN-PR + Prioritized Focused ES has slower learning compared to the the single agent variant in the first 180 episodes. However, as with regular DQN, the single agent variant plateaus in the last step of progression, while the Focused ES version is able to continue progressing towards the optimal policy.

Furthermore, we test if adding more agents to the multiagent variant using DQN +

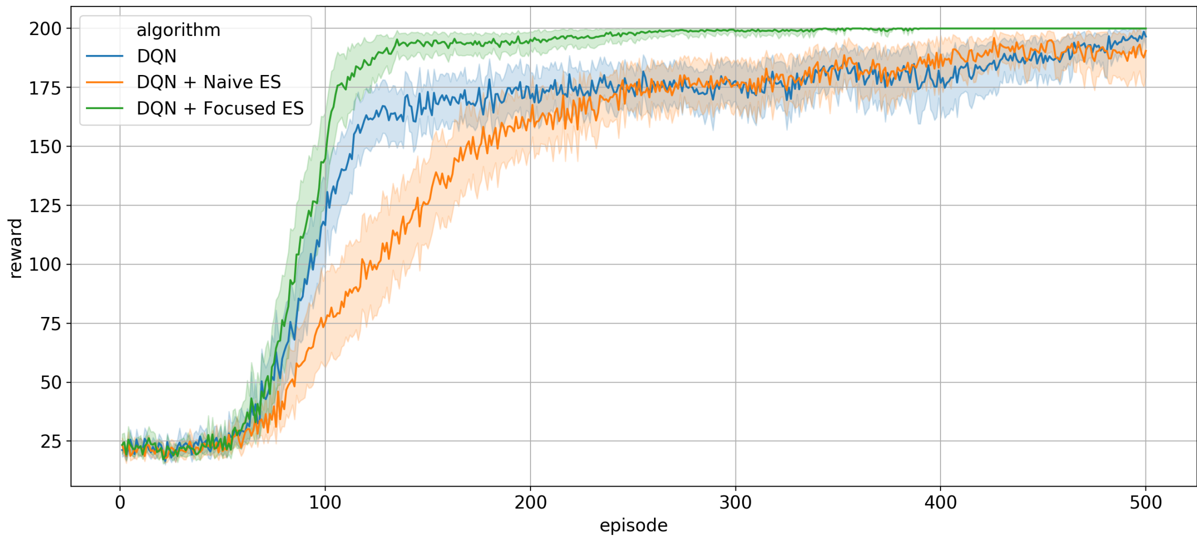


Figure 6.5: Episode reward evolution in DQN.

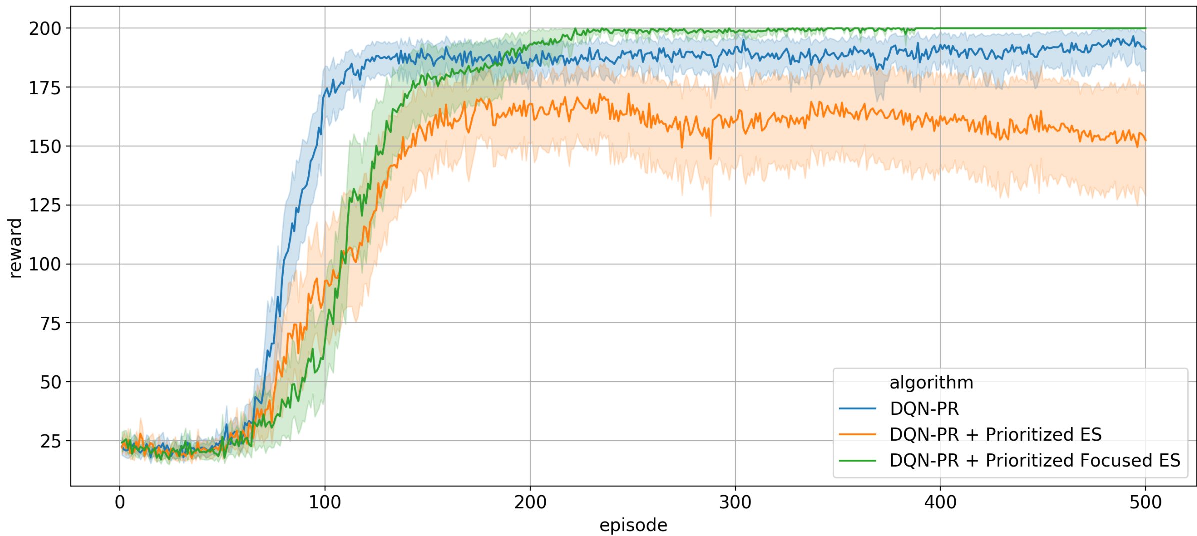


Figure 6.6: Episode reward evolution in DQN-PR.

Focused ES can increase the performance even further. The results are shown in Figure 6.7 and Table 6.7. As expected, the biggest impact occurs when adding a second agent to the experiment. The median, or Q(.5), reduces from 254 with one agent to 111 with two agents.

Increasing the number of cooperative agents further than two yields no significant improvement. There is a small improvement up to four agents reducing the median to 99. A similar behavior is seen for Q(.25) and Q(.75). But after five agents performance starts to decrease. In the ten agents experiment the median is 123, inferior to the two agents experiment. The experiment is conducted only up to ten agents, but the visible trend in Figure 6.7 implies adding more agents might lead to continually decreasing performance.

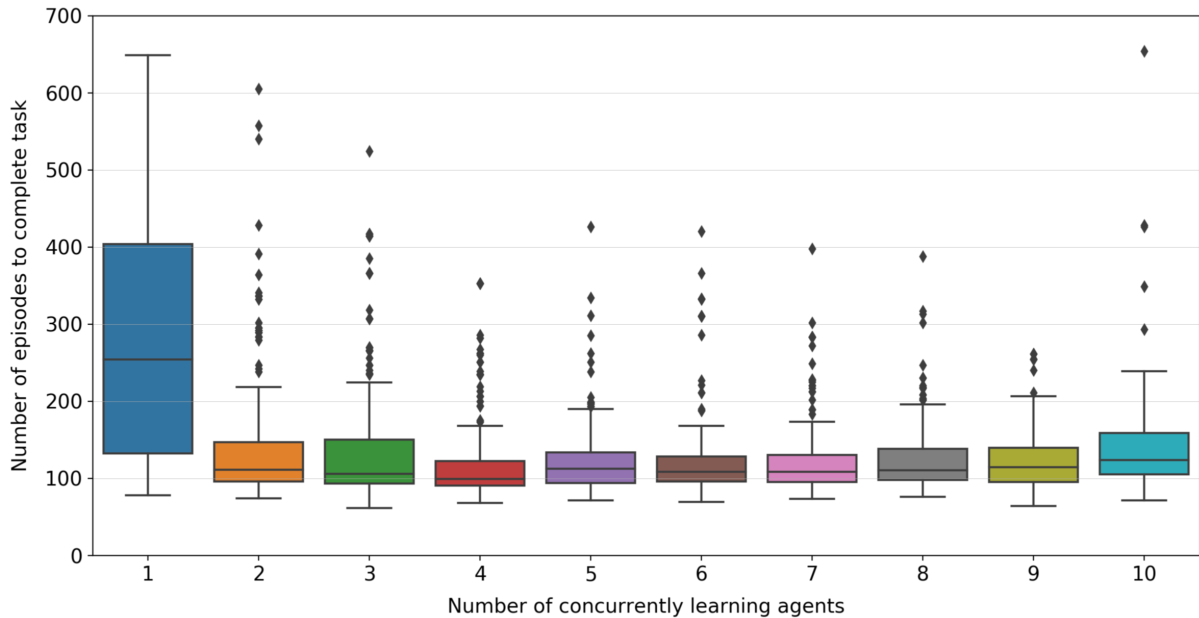


Figure 6.7: Boxplots showing the distribution of number of episodes to complete task with different numbers of concurrently learning agents.

Table 6.2: Q1, Q2 and Q3 ETC for Focused Experience Sharing between 1 to 10 agents.

| Quantile | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Q(.25) | 132 | 96 | 93 | 90 | 94 | 96 | 95 | 98 | 95 | 105 |
| Q(.5) | 254 | 111 | 105 | 99 | 112 | 108 | 108 | 110 | 114 | 123 |
| Q(.75) | 403 | 146 | 149 | 122 | 133 | 128 | 130 | 138 | 139 | 158 |

The last experiment conducted in Cart Pole environment is to see whether the use of tile coding for state discretization in the Focused ES method is able to reduce ETC. In Figure 6.8 we show the results with different tile coding, ranging from 1, which is equivalent to the simple state aggregation, to 41 tiles. The number of tiles were gradually increased until the performance dropped. In Table 6.3 we show the range of offsets applied in each configuration, as well as the ETC mean and deviation.

The results shows that increasing from state aggregation to a small number of tiles has a slight increase in performance measured by ETC mean. The greatest effect though is in reducing ETC standard deviation, which is clearly seen in the density curves plotted. The optimal results are achieved with 11 tiles; the last option, 41 tiles, actually decreases both ETC mean and deviation.

Tile coding is implemented by multiplying the number of occupancy grids by the number of tiles, increasing the amount of space and time to compute. Further evaluation is required to determine if the small improvement in performance achieved by tile coding

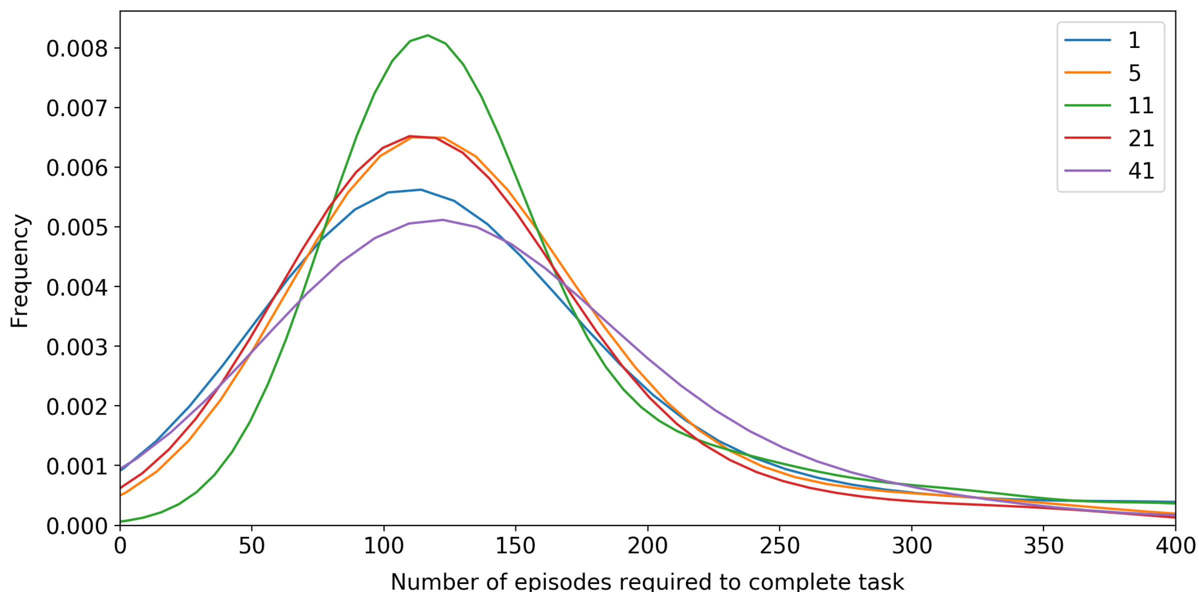


Figure 6.8: Density estimations for different number of tiles used on tile coding applied to Focused ES.

Table 6.3: Comparison of results using different number of tiles for discretization.

| Number of Tiles | Tile Offsets | ETC Mean | ETC Deviation |
|---|---|---|---|
| 1 | - | 157.19 | 130.37 |
| 5 | $\{0.05 \cdot x \text{ for } x \in [-2..2]\}$ | 150.97 | 115.11 |
| 11 | $\{0.02 \cdot x \text{ for } x \in [-5..5]\}$ | 151.70 | 81.32 |
| 21 | $\{0.01 \cdot x \text{ for } x \in [-10..10]\}$ | 151.32 | 120.01 |
| 41 | $\{0.01 \cdot x \text{ for } x \in [-20..20]\}$ | 170.03 | 150.56 |

justifies its use over the simpler state aggregation form of discretization.

## 6.3 Microsoft: Marlo

In this Section, we will review and discuss the experimental results of MACES applied to a RL navigation problem.

### 6.3.1 Environment description

The model was also tested in the Marlo environment, a MARL environment based on platform Malmo (Johnson et al., 2016). Malmo is a platform based on the Minecraft engine, and designed by Microsoft specifically for the purposes of research in RL. Several

researches have been conducted in Malmo since its inception and subsequently in Marlo, and recently a competition was held to evaluate the best performing MARL algorithms in a few selected scenarios (Mohanty, 2018).

We will use the simplest environment called Find The Goal, a navigation problem. In it, the agent starts with in a small room. Its goal is to locate a block in the room, which is randomly initialized in a different position at each episode. The state is represented only by what the agent sees in the screen, a 84x84 single channel image, presented in Figure 6.9. The agent receives a negative 0.01 reward for each action taken, and a 0.5 reward for achieving the goal. The action space is discrete and the agent has four possible actions: turn left or right, or move forward or backwards.

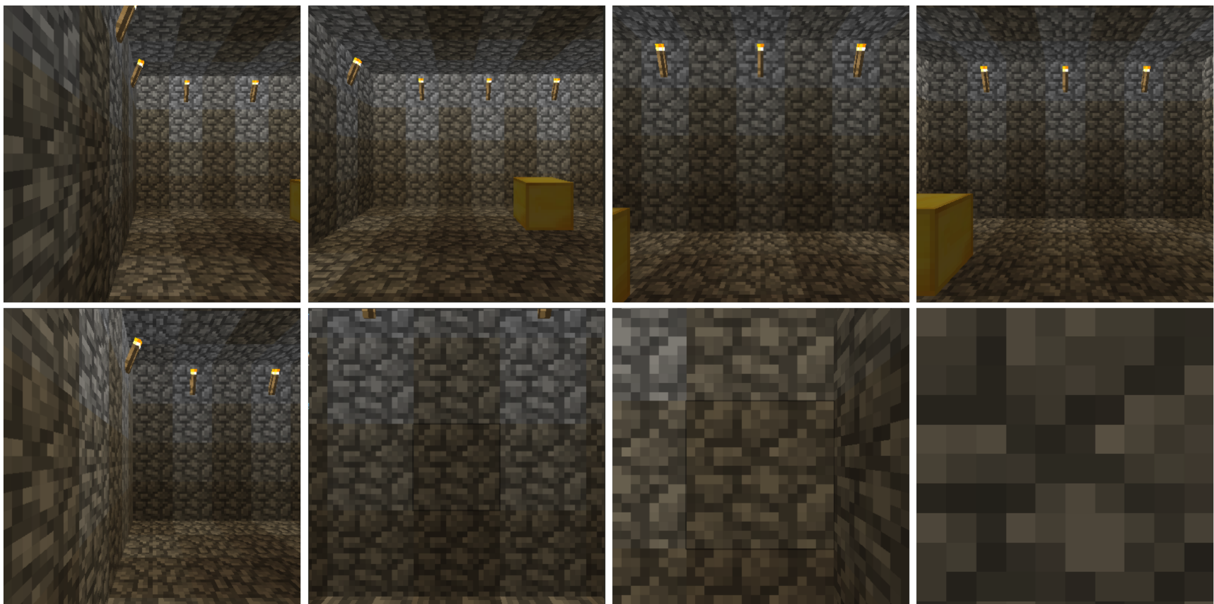

Figure 6.9: Samples of the state perceived by the agent in Marlo environment. These are 84x84 images, here shown in 3 channels (RGB) for clarity. In the top images, you can see the goal, either partially occluded or in full sight. In the bottom row are example of states where the agent cannot see the goal. Notice as the agent approaches the wall the images gets more blurry and the traits are indistinct.

We define the goal as achieving an average reward greater or equal 0.35 in five consecutive episodes. No initial knowledge of the environment is allowed, so learning only initiates after the agent has experienced a number of transitions equal to the size of the replay buffer, which can take between one and two episodes.

As in Cart Pole, the performance of the agent is measured in terms of the number of number of episodes required to complete the task. For this problem, the minimum ETC is 5, as the goal requires a moving average of the last 5 episodes, and the maximum ETC allowed is 100. If the agent is unable to reach the goal within the delimited number of episodes, ETC is set to 100 and the trial is registered as a failure.

### 6.3.2 Results

We will follow the same approach defined for Cart Pole. Due to the significantly longer computational time required in Marlo, caused by the complex state space, only 20 trials were executed for each variant tested. For the same reason, additional experiments with more than two agents and tile codings are also not included for Marlo.

We first compare multiagent variants DQN + Naive ES and DQN + Focused ES with single agent DQN baseline, and multiagent variants DQN-PR + Prioritized ES and DQN-PR + Prioritized Focused ES with single agent DQN-PR baseline. We aim to show that the cooperative multiagent variants can outperform the single agent baseline.

In Figure 6.10 we plot the samples from a single agent DQN versus a multiagent DQN with two agents sharing experiences. In all experiments, a normalized histogram and a kernel density estimation of both distributions are used to compare.

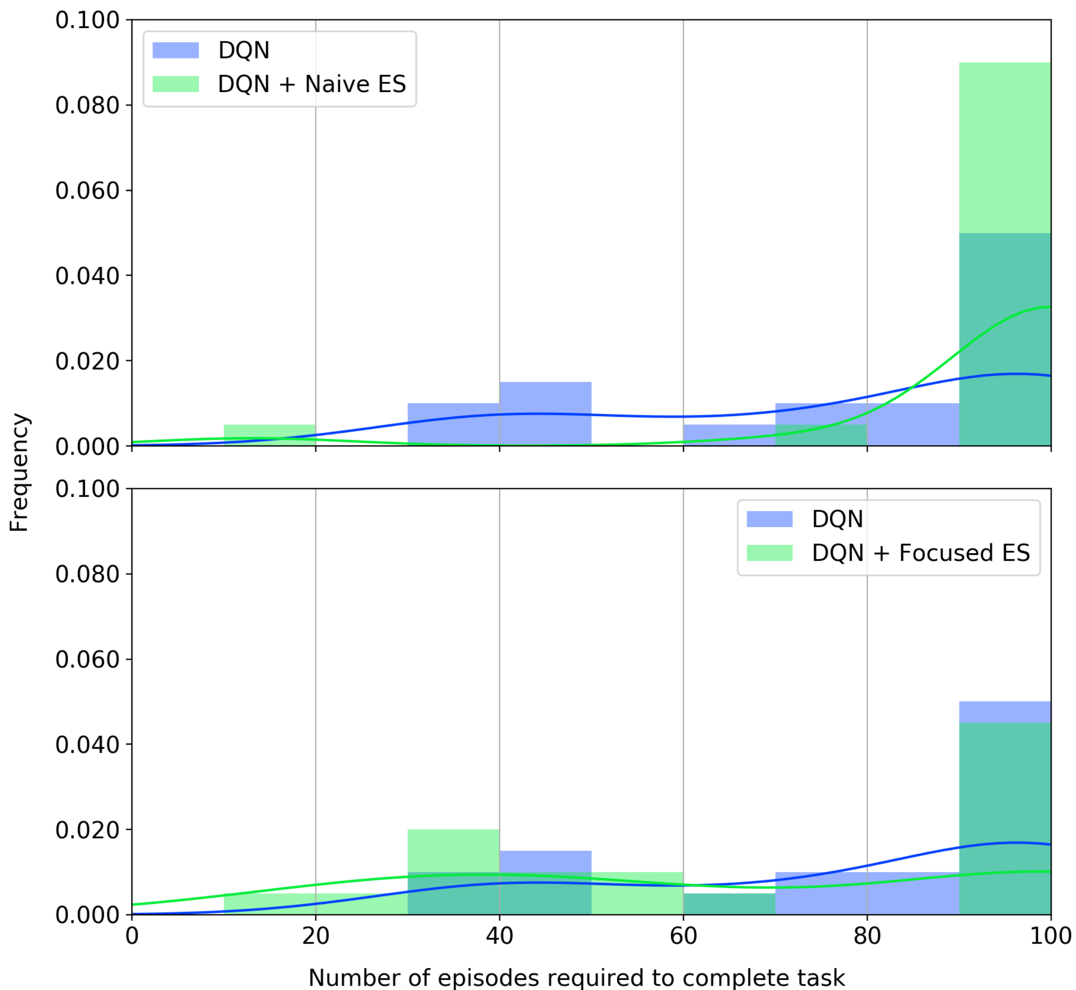


Figure 6.10: Comparison of single agent DQN with multiagent DQN with Naive ES and Focused ES in Marlo environment.

Both single agent and multiagent DQN with Naive ES struggle to complete the task within 100 episodes. DQN + Naive ES has an ETC mean of 94.30, significantly lower compared to single agent ETC mean of 78.75, and it even fails to complete the goal in 18 out of 20 trials. DQN + Focused ES shows a marginally better result, with a 19.75% improvement over single agent DQN, but also stops short of achieving the goal in a similar number of trials. A two sample K-S test fails to reject the null hypothesis that DQN and DQN + Focused ES samples are drawn from the same distribution. The results discussed are presented in Table 6.4.

Table 6.4: Experiments results in Marlo environment.

| Method | ETC Mean | ETC Deviation | Trials Failed | ETC Improvement |
|---|---|---|---|---|
| DQN | 78.75 | 24.55 | 8 | - |
| DQN + Naive ES | 94.30 | 19.56 | 18 | -19.75% |
| DQN + Focused ES | 66.25 | 32.21 | 9 | +15.87% |
| DQN-PR | 10.55 | 5.75 | 0 | - |
| DQN-PR + Prioritized ES | 25.15 | 11.26 | 0 | -138.39% |
| DQN-PR + Prioritized Focused ES | 6.25 | 0.70 | 0 | +40.76% |

The same test is conducted for DQN with Priority Replay, shown in Figure 6.11. DQN-PR is able to achieve the goal in 10.55 episodes on average, a performance far superior than regular DQN. DQN-PR with Naive ES performs far worst than regular DQN, with an increase of 138% in ETC, from 10.55 to 25.15. DQN-PR with focused sharing, on contrary, shows a reduction of 40.76% on the baseline single agent ETC, reducing the mean number of episodes from 10.55 to 6.25, only 1.25 above the minimum required. A two sample K-S test rejects the null hypothesis that DQN-PR and DQN-PR + Prioritized Focused ES samples are drawn from the same distribution, with a p-value of 0.023.

The results achieved in the Marlo environment confirm the results seen in Cart Pole, and hints that the MACES model can be applied to a larger selection of RL with similar success. In Chapter 7, we review past and current research related to our proposed model and review similarities and differences.

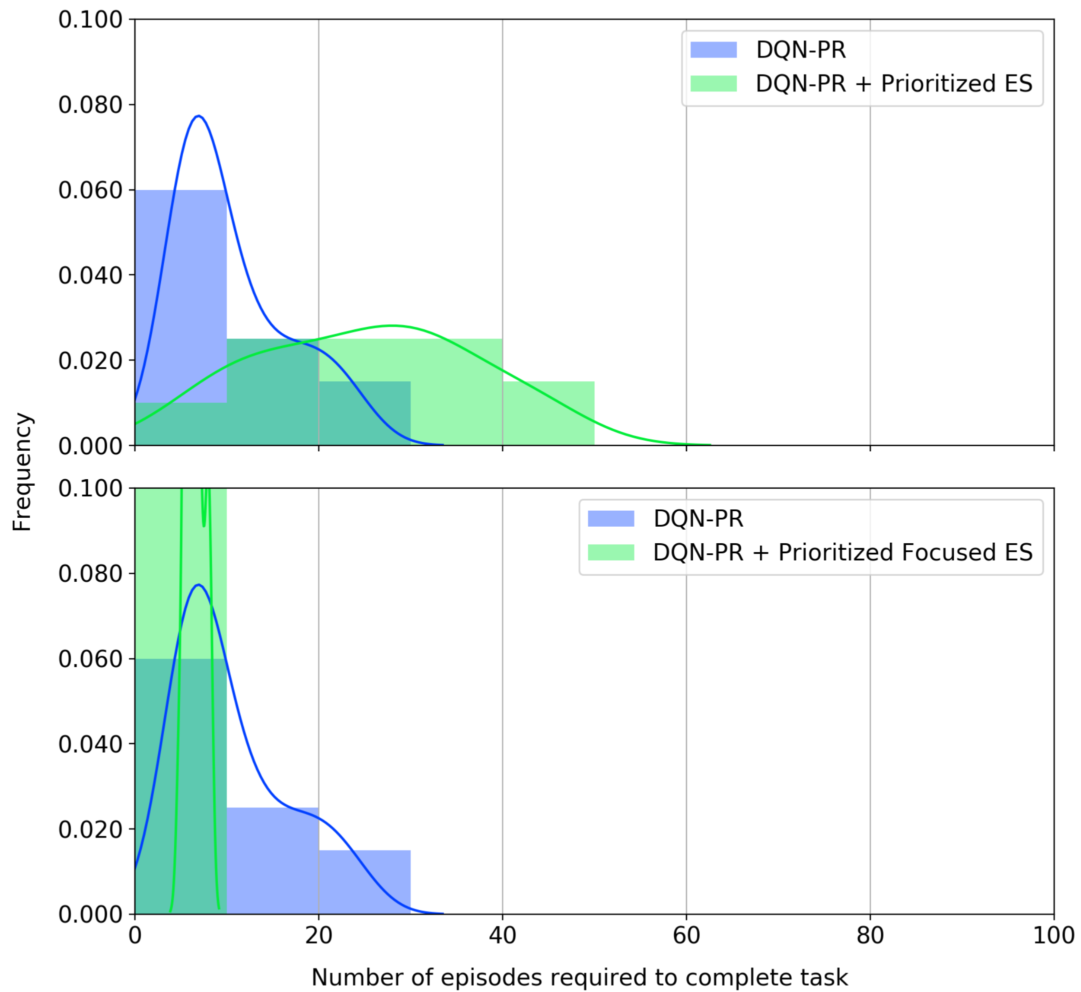


Figure 6.11: Comparison of single agent DQN-PR with multiagent DQN-PR with Prioritized ES and Prioritized Focused ES in Marlo environment. Frequency on the y-axis is cut-off at 0.1 for better visualization.

# Chapter 7

# Related Work

After introducing our proposal, we will proceed to compare it with related existing work. We discuss a wide range of research in related areas to situate our proposal amongst existing state-of-the-art methods, even if the methods are not directly comparable, but anyhow related.

We divide the related work review in three different groups. Section 7.1 discusses proposals related to knowledge sharing through action advice, an approach more common than experience sharing and well explored in the literature. Section 7.2 discusses approaches to extend experience sharing to heterogeneous agents and dynamic environments. Section 7.3 discusses methods to accelerate learning by increasing buffer diversity, from which we draw inspiration to devise the Focused ES method. The work presented includes not only multiagent variants but also single agent with distributed learning.

## 7.1 Sharing action advice

Apart from ES, the main approach to knowledge sharing in MARL has been what is called action advice. Clouse (1996) proposes a model in which the learning agent learns from both the critic, the value function, and a training agent. It introduces the pattern of teacher-student learning in RL algorithms, in which a trainer, an agent who has learned beforehand how to execute a task, indicates to the learner the optimal action to take in a given state.

In the proposed approach, a instruction rate parameter determines the percentage of timesteps to which the trainer instructs the learner. If the trainer does not provide an action, the learner has to choose based on its own policy. In earlier work by Clouse and Utgoff (1992), an agent decides to ask for help when it has low confidence in its own action choices, based on the size of difference of previous state evaluations. The action advice is not directly incorporated into the learner's policy; however, in a value based approach to

RL, receiving a positive feedback from taking an action indicated by a teacher will change the value function and consequently steer the learner's policy toward that action in the future.

Experiments conducted by Clouse and Utgoff (1992) in a race car toy scenario shows that instruction provided by training agents reduce the amount of training necessary to meet the goal, with an optimal instruction rate at about 80%. It is speculated that higher rates could lead the learner to increasingly rely on the trainer and perform less exploration, not allowing the agent to learn from failure.

The action advice method is followed up by Torrey and Taylor (2013), which proposes a knowledge sharing method that could be extended to scenarios where the teacher or student could be either humans or machines. The authors advocate this method because requires minimal similarity between teachers and students. Only action set needs to be the same; state representations might differ, allowing for heterogeneous agents to collaborate.

In their work, Torrey and Taylor (2013) assume the teacher agent can not give unlimited advice. The primary reason for this restriction is that the system should be able to support human teachers, which have limited patience and attention. However, incidentally also limits the cost of communication, a common concern in proposals related to knowledge sharing since a high communication cost can be a hindrance to implement these systems in real-life situations.

Important insights emerge from an empirical evaluation of the methods proposed. Student learning is shown to improve with only a small advice budget, which have a greater impact when it is spent on a few more relevant states. More importantly, they show teaching can improve student learning through action advice even when different learning algorithms or state representations are used, and students can eventually grow to outperform teachers through the methodology.

This work is further extended in da Silva et al. (2017) to include multiagent environments in which all agents can act both as a teacher and a student. The authors point all three situations in which the advice from other agents can be specially useful: when a learning agent joins a system where other agents have been exploring for a while; when a learning agent has not yet been exposed to a particular region of the state space, while other friendly agents have; and when a learning agent internal representation is not as efficient as other agents in a given task. In particular, the situation of an agent not being exposed to a particular region of the state space is key to our proposal of Focused ES.

To coordinate the knowledge sharing between agents, da Silva et al. (2017) propose agents hold a confidence metric based on how many times an agent has explored a desired state. This confidence metric is used by the learner agent to evaluate whether it is necessary to ask for advice, and if advice is asked, it is used by the teacher agent to evaluate

whether it is confident enough to share action advice regarding that particular state. Empirical results in simulated Robot Soccer environment shows learning can be accelerated through this approach even when all agents start the learning process simultaneously, with no previous knowledge.

Although action advice has proven to increase performance in cooperative multiagent settings, it requires instantaneous communication between agents, with an atomic information of one transition shared in each communication action. In our proposal, communication is only done once at the end of each episode, and knowledge regarding several transitions can be packed into a single communication effort, making it more realistically applicable to real world problems.

## 7.2 Sharing with heterogeneity

A significant advantage of action advice over ES is it allows knowledge sharing to occur between agents with different state representations. Methods that alter ES to allow for some heterogeneity in the state space have been proposed and can be a viable solution to extend ES methods to environments with heterogeneous agents or states.

In Verstraeten and Nowé (2018), the main issue of transferring experiences among heterogeneous agents is addressed. The author details an approach to share experiences amongst similar but heterogeneous agents, in what is called fleet applications. The experiments are conducted in a wind farm, where each wind turbine is considered an agent. The agents learn in the same context and have similar state and action spaces. However, small differences in the fabrication process may lead to slight variations in the state or action spaces, hindering the opportunity to freely share experience in the fleet.

A work around is found by discovering which agents are similar enough to benefit from shared knowledge. The solution is a model-based approach, where Gaussian processes are used to model each fleet member's transition function. Gaussian processes are able to model complex non-linear surfaces using limited amount of data. The correlation between the agents are measured through coregionalization, which is used to define which members of the fleet are similar enough, and could eventually be allowed to share experiences (Verstraeten and Nowé, 2018).

Garant et al. (2017) approaches a similar problem. The authors introduce a technique for speeding up multiagent learning by exploiting concurrent and incremental ES, allowing for rapid acquisition of policies in large-scale, stochastic and homogeneous multiagent systems. In this investigation, the agents are homogeneous, but the environments are dynamic. So instead of having to determine agent's similarities, the agents needs to

determine if the environment they are currently in are similar enough to determine if ES will be helpful to its learning process.

The solution proposed introduces a supervisor-directed transfer technique. A supervisor agent constructs a high-level characterization of an agent's non-stationary learning environment, here called contexts, which are used to identify groups of agents operating under approximately similar dynamics in a short temporal window. These agents computes the contextual information for a group of subordinate agents, and groups them by a similarity metric. Agents are only allowed to share experience within these groups, which are constantly recalculated by the supervisors. The context distributions are approximated by a multivariate Gaussian and the Mahalanobis distance is used as the metric. The method, shown in Figure 7.1, is validated on a large network-distributed task allocation problem.

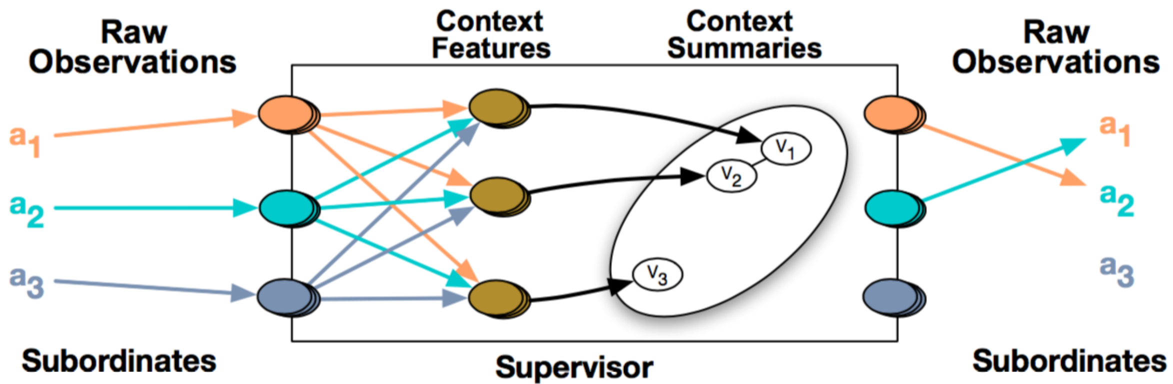


Figure 7.1: Architeture showing how subordinate agents are grouped by the supervisor according to the calculated context features. Reproduced from Garant et al. (2017).

Similarly, plenty research has been conducted covering knowledge sharing between heterogeneous tasks, either by the same agent or different agents. To be able to transfer learned knowledge between tasks is considered the holy grail of ML, a silver bullet that could make existing ML models into actual prospects for artificial general intelligence (Pan and Yang, 2010).

Torrey et al. (2006) discusses an action advice process where the tasks are different. It proposes a human-provided mapping to specify the similarity between source and target task, allowing advice to be given even when the task conducted between agents differ. A similar proposal but extended to model based algorithms is discussed in Taylor et al. (2008). These are further explored in Boutsioukis et al. (2011), where the authors apply intertask mapping specifically to multiagent domains, demonstrating through experiments reduction of learning time. This technique is extended to DRL algorithms in Kong et al.

(2017). The intertask mapping approach could also be a solution to handle heterogeneity between agents and environments in ES models.

## 7.3 Increasing buffer diversity

A lot of research has been conducted in how to increase buffer diversity and whether it leads to better performance, a key tenet behind the Focused ES method. In Nguyen et al. (2018), the authors propose a technique to calculate similarity of new experiences with existing experiences based on a custom distance function. This measure, combined with a stochastic process, is used to determine whether or not to store a new experience. This assures that new experiences are only added if they are different enough from the existing experiences in the buffer. This new technique, combined with PER, shows consistently better results compared to the original DQN.

Related as well is the work conducted in distributed learning using DQN, with similar approaches discussed in Nair et al. (2015) and Ong et al. (2015). Nair et al. (2015) introduces a fully distributed learning algorithm, called Gorilla. In it, several agents learn concurrently, but have a single action-value function and a single replay buffer. The gradient of the loss is calculated locally, and then sent to a centralized network where it is used to update the parameters. Sampling from several environments in parallel leads to increased diversity in the buffer, and the centralized network ensures every learning step is shared among all agents. A representation of the Gorilla framework is seen in Figure 7.2.

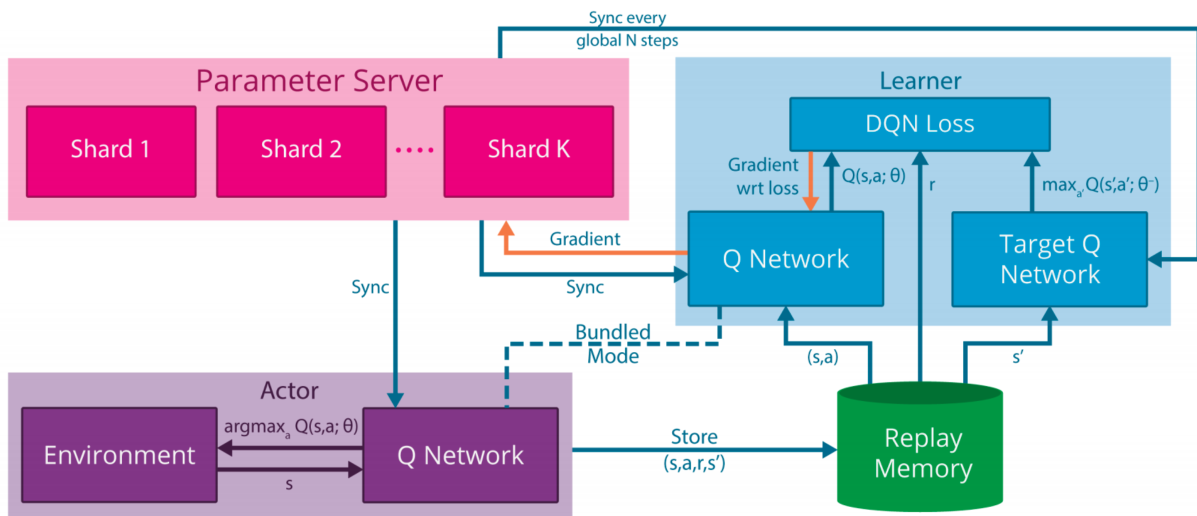


Figure 7.2: Gorilla Framework for distributed learning with multiple learners and actors with a single replay memory and parameter server. Reproduced from Nair et al. (2015).

More recently, in Horgan et al. (2018), the authors argue that having a single replay buffer shared amongst agents is alone sufficient to improve the results of DQN. This statement is investigated empirically and confirmed by the results achieved in the ALE environment. The success is credited to the diverse set of experiences introduced in the buffer, caused mainly by the possibility of having each agent follow different exploration policies in order to form a more diversified buffer. The model, called Apex, is considered an evolution of Gorilla, and is represented in Figure 7.3.

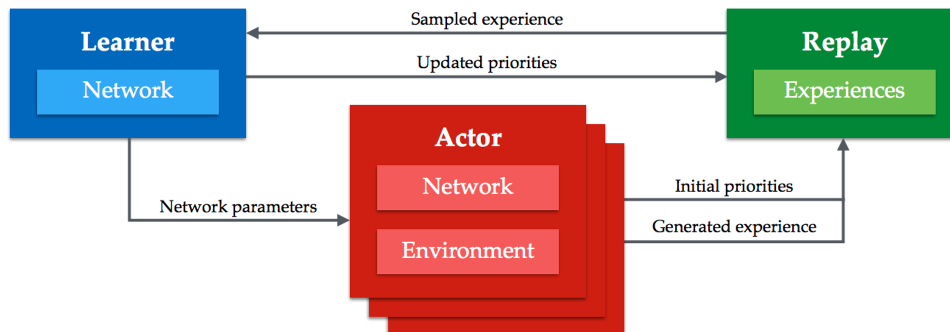


Figure 7.3: Apex Framework for distributed learning with multiple actors, a single learner and a single replay buffer. Reproduced from Horgan et al. (2018).

Our approach differs from this work by considering each agent as an autonomous independent entity, aligned with the MARL problem definition. Focused ES do not require centralized replay buffer or networks, limiting the communication to once per episode during the ES round.

We believe that this makes our approach more easily applicable to real world scenarios, since an unique shared buffer would require constant communication with a central server, which may be impractical in several in industrial applications. In addition, we also propose enhanced ways of deciding which experiences are relevant to share amongst the agents, ensuring that the proper balance of experiences is stored in the buffer that can lead to faster convergence of the learning algorithm.

In Table 7.1 we present an overview of the main related work discussed in this Chapter. For each method we show: (i) whether or not it is applicable to independent learning agents; (ii) what knowledge is shared between agents; (iii) if it is applicable to discrete or continuous state spaces; (iv) which experimental domains were used for validation; (v) the number of methods proposed in the work; (vi) and a brief sentence summarizing its contribution to the field.

In Chapter 8, we conclude with a discuss of some of the limitations of MACES, and interesting directions for future research.

Table 7.1: Related work comparison (in the order they are presented).

| Author | Independent agents | Knowledge shared | State space | Experimental domains | Number of methods | Contribution |
|---|---|---|---|---|---|---|
| Clouse (1996) | Yes | Action advice | Discrete | Race Track | One | Introduces teacher-student framework. |
| Torrey and Taylor (2013) | Yes | Action advice | Discrete, Discretized | Mountain Car, PacMan | Several | Extends teacher-student framework. |
| da Silva et al. (2017) | Yes | Action advice | Discrete, Discretized | Half-field Offense | Several | Extends teacher-student framework, all agents can act both as teacher and student. |
| Tan (1993) | Yes | Experience | Discrete | Hunter-prey | Several | Introduces experience sharing. |
| Verstraeten and Nowé (2018) | Yes | Experience | Continuous | Wind Turbines Fleet | One | Experience sharing between heterogeneous agents. |
| Garant et al. (2017) | Yes | Experience | Discrete | Task Allocation | One | Experience sharing in dynamic environments. |
| Nair et al. (2015) | No | Network parameters | Continuous | Atari 2600 | One | Concurrently learning agents with centralized action-value function. |
| Horgan et al. (2018) | No | Experience | Continuous | Atari 2600 | One | Concurrently learning agents with centralized replay buffer. |
| MACES (2019) | Yes | Experience | Continuous | Cart Pole, Marlo | Several | Improved selection of experiences to share promotes faster learning |

# Chapter 8

# Conclusion

The proposal was validated in two different setups, a classic control environment, Cart Pole, and a more complex navigation problem based on a game engine, Marlo. The very different setups allows us to have a more robust evaluation of the proposed MACES model and its applicability to distinct RL problems.

Our first goal was to answer if naive ES between agents is enough to accelerate multiagent learning between cooperative agents. As the experiments show, naive ES between two agents can either have no impact (Cart Pole) or effectively delay learning (Marlo). The constant flow of unrelated experiences destabilizes learning and it was shown to be harmful for the agent's performance.

Our secondary and main goal was to propose a ES method that can accelerate learning. The model proposed, MACES, allows two agents learning concurrently to learn the same task in half the number of episodes required for non cooperative agents.

The best performing method in MACES is named Focused ES, in which the agents keep track of an occupancy grid showing which regions of the state-action space have already been explored, and only requests experiences related to regions with low exploration. This limits the flow of new experiences to only those that can bring novelty and help the agent increase the reach of its action-value function estimation. Focused ES fixes the major issue found in naive ES, in both regular DQN variant and in DQN-PR, as shown in both Cart Pole and Marlo environments summarized in Tables 6.1 and 6.4. Additional exploration in the CartPole environment shows the method can be further improved by either increasing the number of cooperative agents or using tile coding for state discretization.

We also propose and test a Prioritized ES method, where the agents use TD-error based priorities to define which experiences are more relevant to share, and a combined Prioritized Focused ES method. Prioritized ES alone shows none or little improvement over single agent DQN with Priority Replay, but combined with Focused ES can outperform the single agent variants.

All the experiments were conducted disregarding agents interactions, a key factor in Markov games, the most common formulation of MARL. To extend ER methods to Markov Games we

need to account the non-stationary nature of the environment caused by the other agent's changing policies, and possible effects on the state caused by the agents interactions. We discussed some related work that addresses the issue of applying ER to MARL. An interesting direction of future research would be to extend the Focused ES method to Markov games by combining it with methods that control for the non-stationary nature of MARL, such as either filtering experiences by age or applying a form of importance sampling to decay experiences.

Another relevant limitation is its applicability to only homogeneous environments. Related research were discussed on how to extend ES to settings with either heterogeneous agents or states space. The proposed solution involves determining a measure of similarity to define if the agents or states space are similar enough in order to benefit from ES. As in the intertask mapping proposals, a mapping function can be used to transform experiences and allow experiences between distinct agents to be shared. Extending Focused ES to problems with heterogeneity using the methods discussed can also be an interesting line of research.

It is important to note that Focused ES benefits from exchanges up to ten agents only. A likely scenario that can be considered in future research is a larger number of agents interacting sparsely, requiring an even more careful consideration of which experiences are relevant to share.

Finally, we emphasize MACES do not require a centralized neural network or centralized buffer, and requires limited communication between agents compared to action-advice knowledge sharing methods or centralized learning RL algorithms. This makes it more prone to be applied to environments where the latency and bandwidth of communication between agents are limited. Example applications can include autonomous agents spread over large distance that share experience on episodic basis, such as cooperative autonomous vehicles or manufacturing robots, or autonomous agents that are allowed only to communicate at sparse intervals.

# References


Adam, S., Busoniu, L., and Babuska, R. (2012). Experience replay for real-time reinforcement learning control. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 42(2):201–212. 22

Albrecht, S. V. and Ramamoorthy, S. (2012). Comparative evaluation of mal algorithms in a diverse set of ad hoc team problems. In *Proceedings of the* $11^{th}$ *International Conference on Autonomous Agents and Multiagent Systems*, volume 1, pages 349–356. IFAAMAS. 27

Albrecht, S. V. and Stone, P. (2018). Autonomous agents modelling other agents: A comprehensive survey and open problems. *Artificial Intelligence*, 258:66–95. 26

Altahhan, A. (2018). Td (0)-replay: An efficient model-free planning with full replay. In *International Joint Conference on Neural Networks*, pages 1–7. IEEE. 22

Bellman, R. E. (1957). *Dynamic Programming.* Courier Dover Publications. 7

Boutsioukis, G., Partalas, I., and Vlahavas, I. (2011). Transfer learning in multi-agent reinforcement learning domains. In *European Workshop on Reinforcement Learning*, pages 249–260. Springer. 30, 58

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32. 16

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *CoRR*, abs/1606.01540. 43

Busoniu, L., Babuska, R., and De Schutter, B. (2008). A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, And Cybernetics, Part C*, 38(2). 24, 26, 27, 28, 33

Chincoli, M. and Liotta, A. (2018). Self-learning power control in wireless sensor networks. *Sensors*, 18(2):375. xii, 25

Clouse, J. A. (1996). Learning from an automated training agent. In *Adaptation and Learning in Multiagent Systems.* Springer Verlag. 55

Clouse, J. A. and Utgoff, P. E. (1992). A teaching method for reinforcement learning. In *Proceedings of the Ninth International Workshop on Machine Learning*, ML92, pages 92–101, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. 55, 56

da Silva, F. L., Glatt, R., and Costa, A. H. R. (2017). Simultaneously learning and advising in multiagent reinforcement learning. In *Proc. of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 1100–1108. IFAAMAS. 56

De Hauwere, Y. (2011). *Sparse Interactions in Multi-Agent Reinforcement Learning.* PhD thesis, Ph. d. thesis, Vrije Universiteit Brussel. Cited on. 26

Derthick, M. (1984). Variations on the boltzmann machine learning algorithm. Technical report, Carnegie Mellon, University of Pittsburgh, Dept. of Computer Science. 12

Doucet, A., De Freitas, N., and Gordon, N. (2001). An introduction to sequential monte carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer.

30

Fernandez, J. M. F. and Mahlmann, T. (2018). The dota 2 bot competition. *IEEE Transactions on Games*. 2, 28

Ferster, C. B. and Skinner, B. F. (1957). *Schedules of reinforcement*. Appleton-Century-Crofts, East Norwalk, CT, US. 6

Foerster, J., Nardelli, N., Farquhar, G., Torr, P., Kohli, P., and Whiteson, S. (2017). Stabilising experience replay for deep multi-agent reinforcement learning. In *ICML 2017: Proceedings of the Thirty-Fourth International Conference on Machine Learning*. 26

Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, 9(5):1189–1232. 16

Garant, D., da Silva, B. C., Lesser, V., and Zhang, C. (2017). Context-based concurrent experience sharing in multiagent systems. In *Proc. of the* $16^{th}$ *Conference on Autonomous Agents and MultiAgent Systems*, pages 1544–1546. IFAAMAS. xiii, 57, 58

Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *Deep learning*, volume 1. MIT press Cambridge. 14

Hachiya, H., Akiyama, T., Sugiayma, M., and Peters, J. (2009). Adaptive importance sampling for value function approximation in off-policy reinforcement learning. *Neural Networks*, 22(10):1399–1410. 21

Hearst, M. A., Dumais, S. T., Osuna, E., Platt, J., and Scholkopf, B. (1998). Support vector machines. *IEEE Intelligent Systems and their applications*, 13(4):18–28. 16

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2018a). Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*. 2

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2018b). Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*. 42, 43

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*. 20

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780. 17

Hogg, R. V. and Tanis, E. A. (2009). *Probability and statistical inference*. Pearson Educational International. 42

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., and Silver, D. (2018). Distributed prioritized experience replay. In *International Conference on Learning Representations*. xiii, 60

Hu, J. and Wellman, M. P. (2003). Nash q-learning for general-sum stochastic games. *Journal of machine learning research*, 4:1039–1069. 26

Hu, Y., Gao, Y., and An, B. (2015). Learning in multi-agent systems with sparse interactions by knowledge transfer and game abstraction. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 753–761. IFAAMAS. 26

Johnson, M., Hofmann, K., Hutton, T., and Bignell, D. (2016). The malmo platform for

artificial intelligence experimentation. In *Proceedings of the* $25^{th}$ *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4246–4247. AAAI Press. 28, 50

Kalyanakrishnan, S. and Stone, P. (2007). Batch reinforcement learning in a complex domain. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 94. ACM. 22

Kilner, J. M. and Lemon, R. N. (2013). What we know currently about mirror neurons. *Current biology*, 23(23):R1057–R1062. 26

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980. 38

Kong, X., Xin, B., Wang, Y., and Hua, G. (2017). Collaborative deep reinforcement learning for joint object search. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 58

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems (NIPS)*, pages 1097–1105. 15

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444. xii, 16

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971. 34, 38

Lin, L. J. (1991). Programming robots using reinforcement learning and teaching. In *AAAI*, pages 781–786. 17, 20, 22

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321. 2, 14, 21, 29, 30

Long, J., Shelhamer, E., and Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440. 16

Massey Jr, F. J. (1951). The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78. 43

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133. 15

Mnih, V. (2017). Dqn + variants. `https://sites.google.com/view/deep-rl-bootcamp/lectures`. Accessed: 2018-12-25. xii, 19

Mnih, V., Heess, N., Graves, A., and Kavukcuoglu, K. (2014). Recurrent models of visual attention. In *Advances in neural information processing systems (NIPS)*, pages 2204–2212. 17

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533. xii, 17, 18, 22, 34, 38, 44

Mohanty, S. (2018). Marlo. `https://github.com/crowdAI/marLo`. Accessed: 2018-12-22. 51

Moore, A. W. and Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine learning*, 13(1):103–130. 23

Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., Maria, A. D., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., Legg, S., Mnih, V., Kavukcuoglu, K., and Silver, D. (2015). Massively parallel methods for deep reinforcement learning. *CoRR*, abs/1507.04296. xii, xiii, 22, 59

Nguyen, P. X., Akiyama, T., and Ohashi, H. (2018). Experience filtering for robot navigation using deep reinforcement learning. In *International Conference on Agents and Artificial Intelligence*, pages 243–249. 59

Ong, H. Y., Chavez, K., and Hong, A. (2015). Distributed deep q-learning. *CoRR*, abs/1508.04186. 59

OpenAI (2018). Cartpole environment description. `https://gym.openai.com/envs/CartPole-v0/`. Accessed: 2018-12-10. 43

Palmer, G., Tuyls, K., Bloembergen, D., and Savani, R. (2018). Lenient multi-agent deep reinforcement learning. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 443–451. International Foundation for Autonomous Agents and Multiagent Systems. 26

Pan, S. J. and Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359. 58

Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318. 16

Price, B. and Boutilier, C. (2003). Accelerating reinforcement learning through implicit imitation. *Journal of Artificial Intelligence Research*, 19:569–629. 33

ReNom (2014). Convolutional neural network(cnn). `https://www.renom.jp/notebooks/tutorial/basic_algorithm/convolutional_neural_network/notebook.html`. Accessed: 2018-12-10. xii, 36

Resnick, C., Eldridge, W., Ha, D., Britz, D., Foerster, J., Togelius, J., Cho, K., and Bruna, J. (2018). Pommerman: A multi-agent playground. *CoRR*, abs/1809.07124. 28

RoboCup (2015). Robocup photo gallery. `https://www.robocup.org`. Accessed: 2018-12-25. xii, 28

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition. 1, 4, 5

Salakhutdinov, R. and Hinton, G. (2009). Deep boltzmann machines. In *Proceedings of Machine Learning Research*, volume 5, pages 448–455. 15

Sallab, A. E., Abdou, M., Perot, E., and Yogamani, S. (2017). Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 2017(19):70–76. 18

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized experience replay. *CoRR*, abs/1511.05952. 23, 36, 38

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, 61:85–117. 4, 5, 15, 16, 17

Shalev-Shwartz, S., Shammah, S., and Shashua, A. (2016). Safe, multi-agent, reinforcement learning for autonomous driving. *CoRR*, abs/1610.03295. 29

Shoham, Y., Powers, R., and Grenager, T. (2007). If multi-agent learning is the answer, what is the question? *Artificial Intelligence*, 171(7):365–377. 24

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe,

D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489. 2, 17

Smart, W. D. and Kaelbling, L. P. (2000). Practical reinforcement learning in continuous spaces. In *ICML*, pages 903–910. 22

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in neural information processing systems*, pages 1038–1044. 12

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction.* MIT press. xii, 1, 6, 7, 8, 9, 11, 12, 13, 14, 21, 30, 43

Sutton, R. S., Precup, D., and Singh, S. (1999). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211. xii, 7

Tan, M. (1993). Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proc. of the* $10^{th}$ *International Conference on Machine Learning*, pages 330–337. 2, 29, 30

Taylor, M. E., Jong, N. K., and Stone, P. (2008). Transferring instances for model-based reinforcement learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 488–505. Springer. 58

Tesauro, G. (1995). Td-gammon: A self-teaching backgammon program. In *Applications of Neural Networks*, pages 267–285. Springer. 14

Torrey, L., Shavlik, J., Walker, T., and Maclin, R. (2006). Skill acquisition via transfer learning and advice taking. In *European Conference on Machine Learning*, pages 425–436. Springer. 58

Torrey, L. and Taylor, M. (2013). Teaching on a budget: Agents advising agents in reinforcement learning. In *Proc. of the 12th Conference on Autonomous Agents and MultiAgent Systems*, pages 1053–1060. IFAAMAS. 30, 56

Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100. 20, 38

Vanseijen, H. and Sutton, R. (2015). A deeper look at planning as learning from replay. In *International conference on machine learning*, pages 2314–2322. 22

Verstraeten, T. and Nowé, A. (2018). Reinforcement learning for fleet applications using coregionalized gaussian processes. In *Adaptive Learning Agents (ALA) Workshop at AAMAS).* IFAAMAS. 57

Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J., Schrittwieser, J., Quan, J., Gaffney, S., Petersen, S., Simonyan, K., Schaul, T., van Hasselt, H., Silver, D., Lillicrap, T. P., Calderone, K., Keet, P., Brunasso, A., Lawrence, D., Ekermo, A., Repp, J., and Tsing, R. (2017). Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782. 2, 28

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292. 9

Whitehead, S. D. (1991). A complexity analysis of cooperative mechanisms in reinforcement learning. In *AAAI*, pages 607–613. 2, 29

Wooldridge, M. (2009). *An introduction to multiagent systems.* John Wiley & Sons. 33

Yamins, D. L. and DiCarlo, J. J. (2016). Using goal-driven deep learning models to understand sensory cortex. *Nature neuroscience*, 19(3):356. 17

# Appendix A

# Hyperparameters - Cart Pole

| Hyperparameter | Value | Explanation |
|---|---|---|
| Learning Rate ($\alpha$) | 0.001 | Step-size update for the neural network weights |
| Discount Rate ($\gamma$) | 0.99 | Used to discount future rewards |
| Soft Update Rate ($\tau$) | 0.005 | Step-size update for the target network |
| Experience Buffer Size | 20000 | Maximum number of experiences in the experience buffer |
| Replay Batch Size | 32 | Number of experiences sampled for each learning step |
| Exploration Rate ($\epsilon$) Initial Value | 1.0 | Initial value for $\epsilon$-greedy exploration |
| Exploration Rate ($\epsilon$) Final Value | 0 | Final value for $\epsilon$-greedy exploration |
| Exploration Rate ($\epsilon$) Decay | 4000 | Number of frames over which the initial value of $\epsilon$ is linearly annealed to its final value |
| Experience Transfer Batch Size ($\kappa$) [1] | 128 | Maximum number of experiences shared at each transfer round |
| Priority Replay $\alpha$ [2] | 0.6 | Prioritization exponent, determines how much prioritization is used |
| Priority Replay $\beta$ Initial Value [2] | 0.4 | Initial value of importance sampling correction exponent |
| Priority Replay $\beta$ Final Value [2] | 0 | Final value of importance sampling correction exponent |
| Priority Replay $\beta$ Decay [2] | 10000 | Number of frames over which the initial value of $\beta$ is linearly annealed to its final value |
| Focused ES Threshold ($\zeta$) [3] | 10 | Number of experiences below which the agent considers the region unexplored |

Table A.1: [1]Applied only to multiagent variants.
[2]Applied only to methods with priority replay.
[3]Applied only to methods using Focused ES

# Appendix B

# Hyperparameters - Marlo

| Hyperparameter | Value | Explanation |
|---|---|---|
| Learning Rate ($\alpha$) | 0.002 | Step-size update for the neural network weights |
| Discount Rate ($\gamma$) | 0.97 | Used to discount future rewards |
| Soft Update Rate ($\tau$) | 0.005 | Step-size update for the target network |
| Experience Buffer Size | 10000 | Maximum number of experiences in the experience buffer |
| Replay Batch Size | 128 | Number of experiences sampled for each learning step |
| Exploration Rate ($\epsilon$) Initial Value | 1.0 | Initial value for $\epsilon$-greedy exploration |
| Exploration Rate ($\epsilon$) Final Value | 0.01 | Final value for $\epsilon$-greedy exploration |
| Exploration Rate ($\epsilon$) Decay | 600 | Number of frames over which the initial value of $\epsilon$ is linearly annealed to its final value |
| Experience Transfer Batch Size ($\kappa$) [1] | 128 | Maximum number of experiences shared at each transfer round |
| Priority Replay $\alpha$ [2] | 0.6 | Prioritization exponent, determines how much prioritization is used |
| Priority Replay $\beta$ Initial Value [2] | 0.4 | Initial value of importance sampling correction exponent |
| Priority Replay $\beta$ Final Value [2] | 0 | Final value of importance sampling correction exponent |
| Priority Replay $\beta$ Decay [2] | 10000 | Number of frames over which the initial value of $\beta$ is linearly annealed to its final value |
| Focused ES Threshold ($\zeta$) [3] | 5 | Number of experiences below which the agent considers the region unexplored |

Table B.1: [1]Applied only to multiagent variants.
[2]Applied only to methods with priority replay.
[3]Applied only to methods using Focused ES