Reinforcement Learning and Multi-Agent Systems

Miguel Suau de Castro

Kongens Lyngby 2018

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

We start by presenting the Reinforcement Learning framework from a theoretical point of view. We also study the performance of simple tabular methods of which we outline their main advantages and limitations with some examples of our own.

We then introduce function approximation as a solution for large problems. We present two state-of-the-art deep reinforcement learning methods, PPO and DQN, and compare their performance in more complicated scenarios.

In the last chapter, we explore how to adapt ordinary reinforcement learning techniques to multi-agent systems and show how the game theory concepts of cooperation and competition naturally emerge in these environments.

Inspired by recent work in multi-agent systems, we implement and analyze the capabilities of two new algorithms that try to address the non-stationarity issues in the MDP model, caused by the presence of multiple learning agents. The first method, MADQN, is based on a technique called alternating maximization, while the second, MAPPO, uses a centralized critic to estimate the advantage function.

After testing these algorithms, we discover that traditional methods are still capable of solving the tasks while the time horizon in the generalized advantage estimation is kept long enough. We also show how these perform worse than our multi-agent methods, sometimes even failing to converge, the more the estimates rely on the value function and not on the sample rewards received from the environment.

We finally build a communication channel so as to encourage team collaboration, and prove with two different examples how it can help coordinate actions. The agents learn to communicate with binary arrays that allow them to share information and convey ideas.

All the environments are designed in the Unity Game Engine. Agents are trained with the Unity ML-Agents plug-in, using both our own reinforcement learning algorithms, implemented in Python and Tensorflow, and also the ones provided with the tool.

The algorithms and the rest of source code can be found on GitHub[1] and used for training intelligent agents in Unity. We also produced demonstration videos[2] to show the results of all our reinforcement learning examples.

[1] GitHub: https://github.com/miguelsuau/modified-ml-agents
[2] YouTube: https://www.youtube.com/playlist?list=PLbZouP4AED8BD90wYOffhAO-4UAo65iDy

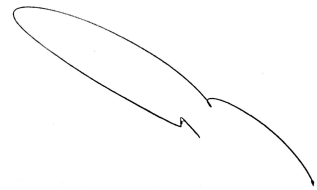
Preface

This thesis is submitted in partial fulfillment of the requirements for acquiring a Master's degree in Mathematical Modeling and Computation at Technical University of Denmark (DTU).

The thesis is prepared at DTU Compute and Unity Technologies, supervised by Lars Kai Hansen, Professor, and Rasmus Larsen, PhD Student, Department of Applied Mathematics and Computer Science, DTU Compute, Technical University of Denmark.

The work of the project was performed in the period from February 22, 2018 and July 22, 2018 and covers 35 ECTS.

Lyngby, 22-July-2018



Miguel Suau de Castro

Acknowledgements

I would like to thank all my colleagues of the Machine Learning team at Unity for creating an amazing tool and making this project possible, and in particular Arthur and Marwan, for their useful advice and technical support on Unity ML-Agents.

I would also like to express my gratitude to my supervisors, Lars and Rasmus, for their valuable guidance and brilliant insights throughout the thesis, but also for their encouragement and inspiration to continue doing research and pursue a PhD after graduating.

I am very grateful to my parents for raising me and giving me the opportunity to be here.

Finally, I would especially like to thank Sofía for providing me with moral and emotional support during these two years in Copenhagen.

Contents

Abstract i

Preface iii

Acknowledgements v

1 Introduction 1
- 1.1 Reinforcement Learning . . . 1
- 1.2 Multi-Agent Systems . . . 2
- 1.3 Unity Game Engine . . . 3
- 1.4 Unity ML-Agents . . . 3

2 Reinforcement Learning 5
- 2.1 Markov Decision Process . . . 6
- 2.2 Partial rewards and expected return . . . 7
- 2.3 Value functions and Bellman equation . . . 7
- 2.4 Static grid world . . . 10
- 2.5 Monte Carlo Methods . . . 11
  - 2.5.1 On-policy MC methods . . . 12
  - 2.5.2 Off-policy MC methods . . . 13
- 2.6 Temporal difference learning . . . 16
  - 2.6.1 On-policy TD methods: Sarsa . . . 17
  - 2.6.2 Off-policy TD methods: Q-Learning . . . 18
  - 2.6.3 Static Grid World with tabular methods . . . 18

3 Deep Reinforcement Learning 21
- 3.1 Value function approximation . . . 21
  - 3.1.1 Loss function . . . 22
  - 3.1.2 Stochastic gradient and semi-gradient descent . . . 23

3.2 Deep Q Network 24
3.2.1 Experience Replay 24
3.2.2 Independent target network 25
3.2.3 Double Q-learning 25
3.2.4 Static Grid World with function approximation 26
3.3 Policy Gradient Methods 28
3.3.1 Policy objective function and gradient 30
3.3.2 MC Policy Gradient: REINFORCE 31
3.3.3 TD Policy Gradient: Actor-Critic 32
3.3.4 Policy Gradient with Baseline 33
3.3.5 Proximal Policy Optimization 35
3.3.6 Dynamic grid world 1: DQN and PPO 37
3.3.7 Dynamic grid world 2: DQN and PPO 42

4 Multi-Agent Reinforcement Learning 45
4.1 Non-Stationary Markov Decision Process 46
4.2 Markov Games 46
4.3 Selecting collective goals and rewards 47
4.4 Centralized and decentralized algorithms 48
4.5 Multi-Agent DQN: Alternating Maximization 49
4.5.1 Multi-Agent grid world: Competition 50
4.6 Multi-Agent PPO: Centralized Critic 53
4.6.1 Multi-Agent grid world: Cooperation 55
4.6.2 Multi-Agent grid world: Learning to communicate 59
4.6.3 Multi-Agent grid world: Learning to collaborate 62

Conclusion and Discussion 67

Appendix 69

Bibliography 80

Chapter 1

Introduction

The combination of dynamic programming algorithms together with supervised learning models and the inspiration of behaviorist psychology have led to important breakthroughs in recent years in the ambit of optimal decision making [1] [2]. Not surprisingly, many research institutions and important technological companies are increasingly focusing more their attention on the field.

Experts in Artificial Intelligence believe that we are now closer to be able to provide machines with the ability to learn, think and act like humans do. Yet, there is a long way to be covered and although there is practical evidence of the capabilities, many questions such as, how to speed up the sometimes prohibitively large learning process or how to accommodate these ideas to real scenarios with multiple interacting agents still remain unanswered.

1.1 Reinforcement Learning

Reinforcement Learning, RL, [3] has primarily been applied to control tasks where an agent, which usually holds very little knowledge of the world, is required to learn an optimal strategy by interacting with its environment. This strategy, normally referred as policy, is nothing else than a function that maps

the agent's observations of the world with the actions to take in order to successfully solve the task. Through out this thesis we will define how the problem is addressed from the mathematical perspective and what the main advantages and weaknesses of this framework are.

The intention is not to break with popular machine learning techniques that have already demonstrated extensive capabilities, but to adapt them so that intelligent agents can learn by themselves with no need for human supervision. In fact, as in many other fields, the recent success in RL was prompt by the use of artificial neural networks [4] and enhanced by the increase in computing power. In this work we present some state-of-the-art deep RL methods and explore them in detail using a variety of examples.

1.2 Multi-Agent Systems

The majority of the work in RL has been directed towards single agent environments. Yet, practical applications usually involve multiple entities working together or competing against each other. In the last part of this thesis we move away from traditional RL methods, which are no longer valid in the multi-agent setting, and present models that do recognize the influence of the other agents in the environment.

Although, the problem of cooperation and competition has been addressed in areas of research like game theory, it is only been applied to abstract sciences such as economy [5] or politics [6]. We discuss how situations of defection or collaboration can also emerge in simpler environments depending on how the problem is formulated [7] [8].

Given that it is difficult to think of a scenario in which the interaction of multiple intelligent agents could negatively affect the outcome of a particular task, this thesis also explores how agents can efficiently share information to optimize team performance [9]. Because communication over a shared network is normally limited, the problem of deciding when and what information should be shared is also given a lot of attention. We implement methods that allow agents to share actions or policies during training but not during execution.

Finally, motivated by how humans convey their ideas and express themselves, we study language as a potential communication method. In a reinforcement leaning setting, language can be substituted by binary arrays representing concepts. An agent can then send a signal to its peers who will process it together with their state observation and act consistently [10] [11].

1.3 Unity Game Engine

Unity is a multi-platform game engine. Developers can use its graphical interface and scripting functionality to design their 2D or 3D games. It also comes with a built-in physics engine that can handle all kinds of physical interactions such as collisions, gravity or accelerations. All these features make the Unity editor very well suited for the design of complex reinforcement learning environments.

A free version of the software for non-commercial use is available for download in Unity's website[1].

All the examples presented in this thesis are made with Unity and using C# to define the dynamics. Scenes, objects and other graphical components in our environments were downloaded from the Unity Asset Store and some of them were modified with Photoshop.

1.4 Unity ML-Agents

ML-Agents is an open source Unity plugin that allows to create reinforcement learning environments and train intelligent agents. It consists of a C# library with a set of functions that can be used to define transitions, rewards, observations or episode termination conditions, and a bridge that connects Unity with the Python API where training occurs.

An implementation of one of the state of the art methods in RL is provided and can be used to train self-built environments. We also wrote three additional algorithms using Tensorflow. These methods are described and assessed using different experiments in the following chapters.

Visit the ML-Agents GitHub repository [2] to find documentation, source code, examples and a installation guide.

[1] https://store.unity.com/download
[2] https://github.com/Unity-Technologies/ml-agents

Chapter 2

Reinforcement Learning

In contrast to other Machine Learning areas such as supervised learning, where we use a labeled dataset to build a function that can map input and output, or unsupervised learning, where the main goal is to find correlations and patterns in the data cloud itself, in reinforcement learning, very little prior information is provided. The data consists of experiences that are continuously collected from the agent-environment interaction. The nature of this process produces a time series of events where samples are very correlated.

Although this sequential dataset can be used to find an optimal solution, the necessity of gathering more information that can better describe the environment makes it difficult to make the common split between training and testing and raises probably the greatest challenge in reinforcement learning, known as exploration-exploitation dilemma.

In this chapter we will introduce the reinforcement learning framework, describe some of the most popular techniques that intend to tackle the decision making problem and show some examples so as to illustrate the advantages but also the limitations of these methods.

The theoretical content of this chapter is based on [3]. The reader is referred to chapters 3, 4, 5, 6 and 7 in this book for a detailed explanation of all the concepts introduced here.

2.1 Markov Decision Process

In order to address the reinforcement learning problem from a mathematical perspective the agent-environment interaction, figure 2.1, is represented as a Markov decision process MDP. In every time step $t = 0, 1, 2, ..., T$, the agent, using the information provided by the environment about the current state S_t, selects an action A_t that will lead it with certain probability to the next state S_{t+1}, and obtains a reward R_{t+1} as outcome of the chosen action. Eventually, if the process is finite, the agent will reach the terminal state and the sequence will conclude. In a finite MDP, the time series of events from the initial state S_0 to the terminal state S_T is known as episode and must always occur as follows:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2..., R_T, S_T. \tag{2.1}$$

Because the problem is formulated as an MDP each state and reward in an episode is only dependent on the immediately preceding state and action pair. Therefore, in a non-deterministic environment the joint probability of a obtaining a specific reward and ending in a particular state can be written as

$$p(s', r|s, a) = Pr\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\}, \tag{2.2}$$

where s, s' and a are certain states and actions that lie within the complete set of possible states $\mathcal{S}$ and actions $\mathcal{A}$ respectively. Besides, since initial state and terminal state in the previous episode are completely uncorrelated, the first one follows its own independent distribution.

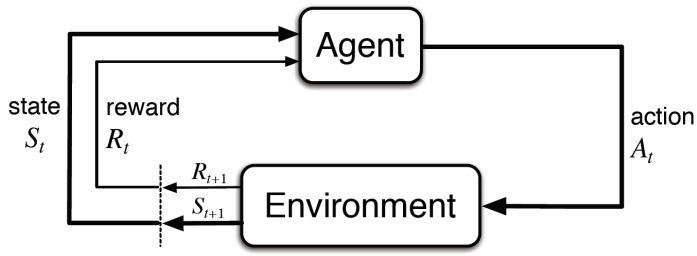


Figure 2.1: Agent-environment interaction [3]. In every step the agent selects an action according to some state information coming from the environment, and receives a reward signal as a consequence of the action taken.

2.2 Partial rewards and expected return

A reward is a a feedback signal that the agent receives at every time step as an indicator of how beneficial is to take a particular action. Negative rewards are meant to penalize a decision, while positive ones encourage a particular behavior. Although, these rewards can be given by the environment, e.g. score in a video game, in certain applications they need to be carefully selected.

The overall objective in reinforcement learning is to maximize the expected return, that is, the sum of expected rewards in a episode. Therefore partial rewards must be well aligned with the final goal of the agent. A more general formulation which also considers the decaying importance of future rewards at the current time step is given by

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k. \tag{2.3}$$

The expression above is known as expected discounted return and the parameter $\gamma \in [0, 1]$ is the discount rate. Thus, for γ close to 1 more weight is given to future rewards, while when γ is small or 0 only proximal or immediate rewards are considered. Note that equation 2.3 is also suitable for continuing tasks, that is, when the process cannot be split into episodes, in that case $T = \infty$ and $\gamma < 1$ for the expected return to be bounded.

2.3 Value functions and Bellman equation

A common strategy to solve the reinforcement learning problem is to learn a value function, which maps every possible state in the environment with its expected return when following a certain behavior. This behavior, which is known in the literature as policy and is denoted by π, represents the agent's probability of selecting a particular action when in a certain state, $\pi(a|s)$. Using this concept we can write the state-value function under policy π as

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \;\middle|\; S_t = s \right]. \tag{2.4}$$

The action-value function 2.5, on the other hand, not only evaluates the value of

being in a state but also the effect of choosing a particular action and following a given policy π from that point onwards:

$$q_\pi(s,a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \;\middle|\; S_t = s, A_t = a \right]. \tag{2.5}$$

Equation 2.4 can be expressed recursively in terms of subsequent states by exploiting some of the properties of MDPs:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s \right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s' \right] \right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma v_\pi(s') \right]. \end{aligned} \tag{2.6}$$

As mentioned before, the goal of the agent is to maximize the expected return over a sequence of steps. Assuming that the true action-value function is known for every state an action, the problem could be easily solved by consistently selecting the action that maximizes the expected return in every state. In that case we say the agent is following the optimal policy. Consequently, the return of any other policy must be always smaller or equal to that of the optimal policy:

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.7}$$

and the optimal action-value function

$$q_*(s,a) = \max_\pi q_\pi(s,a). \tag{2.8}$$

Therefore, considering that our policy is being selected so that the return is

maximized the following condition must always hold:

$$\begin{aligned} v_*(s) &= \max_a q_*(s,a) \\ &= \max_a \mathbb{E}_* \left[G_t | S_t = s, A_t = a \right] \\ &= \max_a \mathbb{E}_* \left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a \right] \\ &= \max_a \mathbb{E}_* \left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a \right] \\ &= \max_a \sum_{s',r} p(s', r|a, s) \left[r + \gamma v_*(s') \right] \end{aligned} \tag{2.9}$$

which is known as Bellman optimality equation for the state-value function. Analogously we can write the optimality equation for the action-value function as

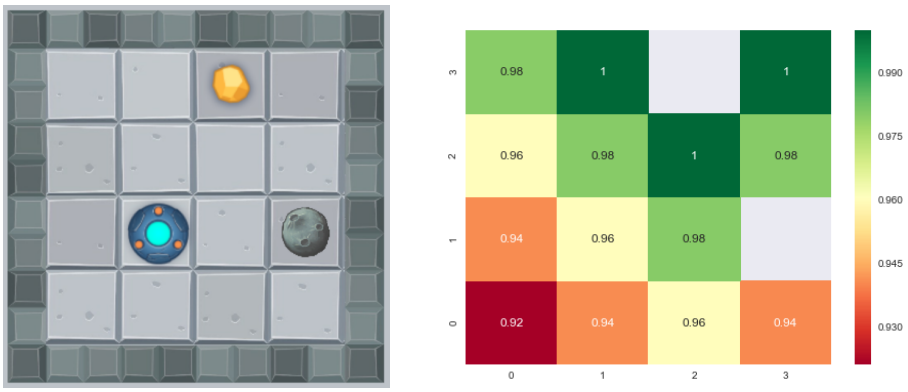
$$\begin{aligned} q_*(s,a) &= \mathbb{E}_* \left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \Big| S_t = s, A_t = a \right] \\ &= \sum_{s',r} p(s', r|a, s) \left[r + \gamma \max_{a'} q_*(s', a') \right]. \end{aligned} \tag{2.10}$$

The exact solution of the above expressions can be found if the state space is finite and the dynamics of the environment known. Thereupon, the best policy can be found by acting greedily in every state from the beginning until the end of the episode. Note that the greedy action in every state is not necessarily the one that maximizes the immediate reward but the final return, which is given at every state by the Bellman optimality equation.

There exist efficient dynamic programming algorithms that can be used to find the optimal policy of a finite state-space environment where the dynamics are known. Although all these methods ensure convergence they are computationally very demanding and only work well in practice when the total number of possible states is small. A more detailed description of these methods can be found in [12] and [3].

2.4 Static grid world

In this section we are going to introduce one of the sample environments that will be used throughout this thesis to explain the theory and evaluate the performance of our reinforcement learning algorithms.



(a) (b)

Figure 2.2: 4×4 static grid world. The blue agent, which can take four different actions, moving north, south, east or west, must collect the gold nugget and avoid colliding with the asteroid (a). State-value function of deterministic grid world when following an optimal policy. Q-learning finds the exact value of equations 2.4 at every state-action pair when following the optimal policy and setting $\gamma = 0.99$ (b)

The blue agent located at the center of the 4×4 grid in figure 2.2 (a) is an UFO that is traveling around the space collecting gold nuggets and avoiding a collision with the asteroids. The UFO must decide in every step between four different actions: moving north, south, east or west. In our first example, the location of the objects, asteroid and gold nugget, is fixed, and thus, the entire set of possible states $\mathcal{S}$ can be defined only using the position of the UFO. The agent receives a reward of $+1$ if the gold nugget is collected, a reward of -1 when colliding with the Asteroid and a reward of -0.01 for every step until the episode is over. An episode finishes when either the agent crashes into the asteroid or the gold nugget is picked up.

We solved this problem using a technique called Q-learning that we describe in section 2.6.2. However, since the dynamics of the environment are known and the state space is finite, it can be solved using Bellman's equation and a dynamic programming method. The color map in figure 2.2 (b) shows the return at every

state when following the optimal policy.

2.5 Monte Carlo Methods

As mentioned before, dynamic programming algorithms require complete knowledge of the environment and the probability distributions of its transitions. The methods described in this section try to approximate the exact solution using simulations. These follow an empiric strategy based on experiences gathered by the agent when interacting with the environment.

The Monte Carlo algorithm (MC) for evaluating a given policy estimates the value function at every state by averaging the sample returns over various trajectories. In this method the value of all the states visited in the episode is updated only at the end of the episode. Depending on whether only the first visit or all the visits to the same state during an episode are used to calculate the average return, two algorithms with different theoretical properties are obtained. It can be proved that both methods converge to the optimal solution as the number of visits to all the states goes to infinity [13]. The pseudo-code below corresponds to first-visit MC.

```
Define: policy π to evaluate
Initialize value function: V
Allocate memory: states, rewards, returns
repeat
    Sample initial state: s
    while not done do
        action ← choose an action following π
        s, r, done ← take step
        states, rewards ← append sr
    end
    for each different s in states do
        G ← sum rewards from first instance of state to end of episode
        returns(s) ← append G
        V(s) ← average returns(s)
    end
end
```

Algorithm 1: First visit MC method for policy evaluation

In this and the following algorithms we will use the boolean variable *done* to indicate whether or not the episode has ended.

The method can be easily modified to calculate q_π instead of v_π. However, these methods can only be used to evaluate a certain policy but not to find the optimal one. One way to do so, would be to update the policy as new information about the value of every state and action comes in. This method is known in dynamic programing as generalized policy iteration (GPI).

The idea is to start with an arbitrary policy, find the value function, select the greedy policy that maximizes the current value function and repeat the process again as shown in the diagram below 2.3. A policy can be updated after a few iterations of policy evaluation even if the exact value function for the previous policy has not been found yet. This strategy is meant to avoid the unnecessary computations of evaluating a sub-optimal policy. In practice a policy can be updated right after every episode, in this case it is common to use experiences coming from different policies to calculate the average return of every state. The main drawback of this method is that even if the process went on forever we could not assure that all states and actions would be explored infinitely many times to be able to guarantee convergence to the optimal policy.

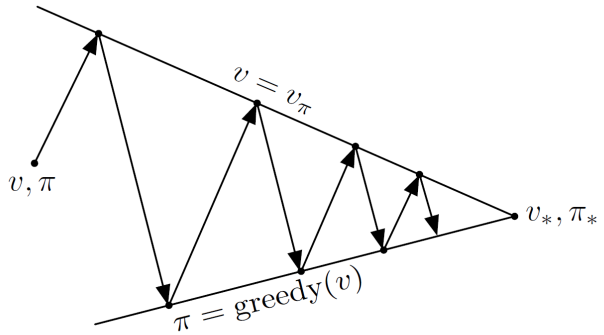


Figure 2.3: Generalized Policy Iteration [3]: Starting from any random policy, the optimal policy is obtained by repeatedly calculating the value function and then selecting the greedy policy that maximizes the return at every state, according to the current value estimates.

2.5.1 On-policy MC methods

On policy methods attempt to guarantee that all regions in the state-action space are sufficiently visited by initially deploying a more relaxed policy where the probability of selecting any of the actions in a given state is greater than

0. One such policy is called ϵ-greedy and the parameter ϵ represents the chance of selecting a random action instead of the greedy one. Hence, combining this strategy with policy evaluation and policy improvement we obtain an algorithm that can be shown to converge to the optimal solution.

Randomly Initialize: π
Initialize action value function: Q
Allocate memory: states, rewards, returns
repeat
  Sample initial state: s
  while *not done* **do**
    $a_{max} \leftarrow \max_a Q(s, a)$
    $a \leftarrow$ choose a_{max} or random a with probability ϵ and $1 - \epsilon$
    $s', r, done \leftarrow$ take step
    states, actions, rewards $\leftarrow$ append s, a, r
    $s \leftarrow s'$
  end
  for *each different* $s - a$ *pair in states-actions* **do**
    G $\leftarrow$ sum rewards from first instance of state-action to end of episode
    returns$(s, a) \leftarrow$ append G
    $Q(s, a) \leftarrow$ average returns(s, a)
  end
end

Algorithm 2: On-policy ϵ-greedy MC method for control

2.5.2 Off-policy MC methods

Although in the previous section we introduced soft-policies to allow some environment exploration instead of acting greedily in every iteration, this had the consequence of making the method learn from a policy that was not entirely optimal. We will now present a different approach where two policies are used instead. The first one, known as behavior policy, is meant to explore and add new experiences, while the other, the target policy, uses the information provided by the first one to make improvements until converging to an optimal strategy.

A key concept in off-policy algorithms is importance sampling [14]. This is used to correct for the fact that the data generated to update our target policy comes from a different distribution.

The probability of a sequence of events occurring when following a specific policy

π can be calculated as

$$P(A_t, S_{t+1}, A_{t+1}, ..., S_T | S_t, A_{t:T-1}\ \pi) = \prod_{k=t}^{T-1} \pi(A_k|S_k) p(S_{k+1}|S_k, A_k) \quad (2.11)$$

and the importance sampling ratio, which measures the relative probability that a trajectory occurs when following target and behavior policies, is given by:

$$\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k) p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k) p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}. \quad (2.12)$$

Note that transition probabilities in numerator and denominator cancel out as they are equal and independent of the policy being followed.

The ratio in equation 2.12 can be interpret as a correction factor and used to estimate the returns of the target policy by using samples obtained with the behavior policy:

$$V(s) = \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathcal{T}(s)|}, \quad (2.13)$$

$$V(s) = \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1}}, \quad (2.14)$$

with $\mathcal{T}(s)$ being the set of all time steps in which state s is visited. Equations 2.13 and 2.14 are the ordinary and weighted importance sampling estimators of $V(S)$ respectively. A detailed description of how these two equations relate to each other can be found in [3].

Algorithm 1 can be then adapted to evaluate a target policy using a behavioral

policy to generate new experiences.

Initialize target policy π
Define behavioral soft policy b
Initialize action value function: Q
Set $G = 0$, $W = 1$
Allocate memory: states, rewards, returns
repeat
  Sample initial state: s
  while *not done* **do**
    action $\leftarrow$ choose an action following b
    s', r, done $\leftarrow$ take step
    states, actions, rewards $\leftarrow$ append s, a, r
    $s \leftarrow s'$
  end
  for *each* $\langle s, a, r \rangle$ *tuple from terminal to initial state* **do**
    $G \leftarrow \gamma G + r$
    $C(s,a) \leftarrow C(s,a) + W$
    $Q(s,a) \leftarrow Q(s,a) + \frac{W}{C(s,a)}[G - Q(s,a)]$
    $\pi(s) \leftarrow \arg\max_{a^* \in \mathcal{A}} Q(s, a^*)$
    $W \leftarrow W \frac{\pi(a|s)}{b(a|s)}$
    if $W = 0$ **then**
      break
    end
  end
end

Algorithm 3: Off-policy MC control method. Estimating optimal policy $\pi \approx \pi^*$

Where W in 3 is a variable holding the importance sampling weights at every step of the episode and $C(S_t, A_t)$ is the cumulative sum of weights in each particular state-action pair.

On the other hand, the expression used for the action-value function update,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)], \tag{2.15}$$

is equivalent to equation 2.14 and allows to reestimate the expected return incrementally as new information comes in.

2.6 Temporal difference learning

Even though, as discussed in the previous section, Monte Carlo methods are proved to converge to the optimal solution they make an inadequate use of the information and can perform poorly in practice. The fact that we need to wait until the end of an episode to make a change in our value function can have a strong negative impact specially in situations where episodes are long or even infinite.

While MC methods use the true sample return obtained at the end of an episode to update the value estimate of all the states and actions visited in that episode, temporal difference learning methods, TD, base the updates on previous estimations of the value function at subsequent states:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right], \tag{2.16}$$

where $R_{t+1} + \gamma V(S_{t+1})$, instead of G_t, is used as the approximation of the expected return at S_t. The TD learning alternative to algorithm 1 used for policy evaluation is outlined below.

Define: policy π to evaluate
Initialize value function : V
repeat
  Sample initial state: s
  while *not done* **do**
    a ← choose an action following π
    s', r, done ← take step
    $V(s) \leftarrow V(s) + \alpha \left[r + \gamma V(s') - V(s)\right]$
    $s \leftarrow s'$
  end
end

Algorithm 4: TD learning for policy evaluation

The technique applied in algorithm 4 is known as bootstrapping and it allows to perform updates more frequently than MC methods while still converging to the optimal solution. In practice, the efficient use of data and memory in TD learning methods, normally leads to faster learning. However, there is no mathematical prove supporting this empirical fact at the moment [3].

Although the methods presented in this and the following sections are based solely on the reward obtained after a single step, waiting a few more steps to do the update normally leads to faster convergence. This is the idea used by

n-step TD methods which intend to combine the low bias high variance estimate of MC methods with the high bias low variance 1-step TD update. The value of S_t is updated after n steps using the intermediate rewards and the current estimate of the return at S_{t+n}.

2.6.1 On-policy TD methods: Sarsa

The on-policy version of TD control algorithms is known as Sarsa. As in MC we also apply Generalized Policy Iteration to evaluate and improve the action choices but in this case the value function update is done online and not at the end of every episode.

Again, the goal is to estimate the action value function $Q(s, a)$ at every state-action pair so as to be able to make decisions on what actions to take, by comparing the expected returns in a particular state. Because this is an on-policy method we need to deploy soft policies in order to allow some exploration around the state-action space. Algorithm 5 presents the Sarsa TD method for control with ϵ-greedy policy

Randomly Initialize policy: π
Initialize action value function: Q
repeat
  Sample initial state: s
  $a_{max} \leftarrow \max_a Q(s, a)$
  $a \leftarrow$ choose a_{max} or random a with probability ϵ and $1 - \epsilon$
  while *not done* **do**
    s', r, done $\leftarrow$ take step
    $a'_{max} \leftarrow \max_{a'} Q(s', a')$
    $a' \leftarrow$ choose a'_{max} or random a' with probability ϵ and $1 - \epsilon$
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma Q(s', a') - Q(s, a) \right]$
    $s \leftarrow s', \quad a \leftarrow a'$
  end
end

Algorithm 5: On-policy TD method for control: Sarsa

Algorithm 5 is guaranteed to converge to the optimal policy if all the state-action pairs are visited infinite times and the soft policy gradually becomes the greedy policy.

2.6.2 Off-policy TD methods: Q-Learning

Q-learning [15] appears as a simple but interesting off-policy variant of Sarsa. In this method the value function is updated after a single step using the immediate reward and the current estimate of the greedy action at the next state, that is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(S_t, A_t) \right] \tag{2.17}$$

Randomly Initialize policy: π
Initialize action value function: Q
repeat
  Sample initial state: s
  while *not done* **do**
    $a_{max} \leftarrow \max_a Q(s, a)$
    $a \leftarrow$ choose a_{max} or random a with probability ϵ and $1 - \epsilon$
    $s', r, done \leftarrow$ take step
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
    $s \leftarrow s'$
  end
end

Algorithm 6: Off-policy TD method for control: Qlearning

Q-learning is considered an off-policy method because the policy being followed is different from the greedy policy used to update the action value function. Yet, algorithm 6 varies from the off-policy methods described in 2.5.1 in the sense that target and behavioral policies are the same and the update is performed using the next state's greedy action. Therefore, the policy must be soft to ensure some exploration.

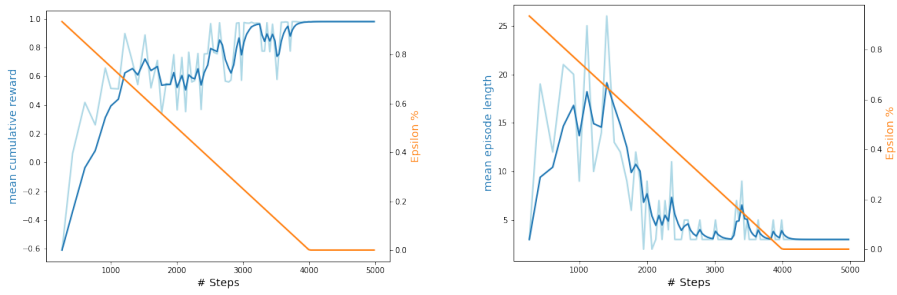
The method works very well in practice and has been proved to converge if all state-action pairs are visited enough times. Another variant, called Expected Sarsa, uses the expectation of the action value function at the next state to make the update.

2.6.3 Static Grid World with tabular methods

Algorithm 6 is implemented in Python and tested on our grid world example. The action-value function $Q(s, a)$ consists of a lookup table where every row represents a state and every column an action. Since the objects always remain

in the same position, states are completely determined by the position of the UFO. Consequently, the size of the lookup table is 16×4 because there are 16 cells in the grid and 4 possible actions.

Since we assume no prior knowledge about the position of the objects all state-action pairs are initialized to 0. To encourage the UFO to explore the environment we plan a linear decay for ϵ from 1, entirely random policy, to 0, greedy policy. The results are shown in figure 2.4



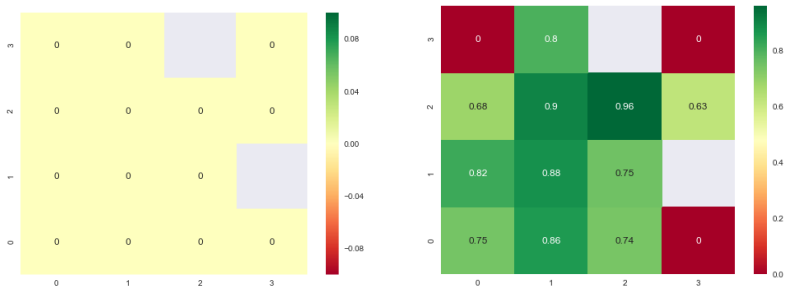
(a) **(b)**

Figure 2.4: Q learning with lookup table. Figure a: The left vertical axis shows the sum of rewards per episode averaged over 10 episodes. Figure b: The left vertical axis shows the total number of steps per episode averaged over 10 episodes. The right vertical axis in both plots corresponds to the value of ϵ at every step. We have applied exponential smoothing to the two blue curves in order to attenuate the peaks. The light blue curves are the true values.

Figure (a) shows that, as the agent learns more about the value function and the chance of random action decays, the UFO begins to consistently collect the gold nugget and avoids colliding with the asteroid. We can see that after 4000 steps the agent starts to act always greedily. This has an obvious effect on the returns which from that point onwards lie always close to 1. The mean episode length depicted in figure 2.4 plot (b) shows that the episodes are short at first. The UFO is moving randomly which means that it can easily either collide with an asteroid or unintentionally collect the gold nugget. Around 1000 steps the agent has learned to avoid the asteroids and thus, the episodes are longer. From step 1500 onwards the agent tries to find the optimal path so that it can solve the task in the fewest possible number of moves.

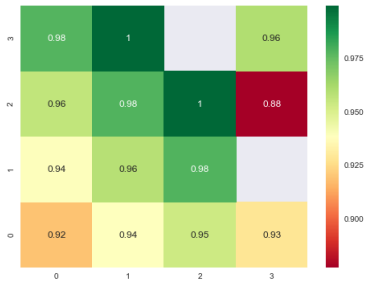
Figure 2.5 shows the estimated action-value of every position after 10, 100, 500, 1000 steps. The values correspond to the action that maximizes the return at each location.

The color map demonstrates how after very few iterations the agent is able to find an accurate estimate of the action-value function and use it to improve its policy. However, one can argue that the method works very well due to its over-simplistic complexity and that using a table to represent the action-state space is not always possible. In the next section we shall study methods with better scaling properties that use functions instead of tables that allow the agents learn in more complex environments.

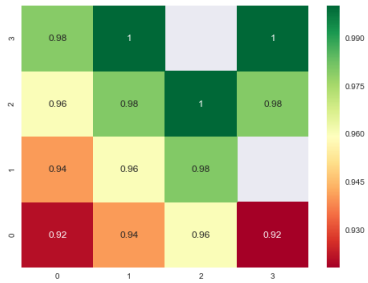


(a) 10 steps

(b) 100 steps



(c) 500 steps



(d) 1000 steps

Figure 2.5: Action-value function estimation using Q-learning with tabular methods. The values shown in the color map correspond to the greedy action at every state.

The algorithm and rest of the source code are available on GitHub[1]. The tutorial on RL with Tensorflow [16] was a valuable reference for the implementation of this method.

[1] https://github.com/miguelsuau/modified-ml-agents

Chapter 3

Deep Reinforcement Learning

As shown in the static grid world example, tabular methods are very useful for solving simple tasks where the number of states and actions is relatively small. Recall that in the example, there were only 16 different states and 4 actions. Nonetheless, if we think of any real control environment the number of possible states and actions can be tremendously large or even infinite. That is why, using a table to represent states and actions is not always possible, because it does not scale when moving to more complex settings.

In this chapter we will present a solution to this problem that implies a dramatic change in the way we apply and optimize our value function. Yet, the reinforcement learning framework remains unchanged and all the algorithms we already presented can still be applied.

3.1 Value function approximation

As opposed to tabular methods, the value function is approximated using a parametric function instead of a matrix where the number of parameters is often smaller than the number of state-action pairs. The function used to represent the

environment depends very much on its complexity. The range of possible options go from a simple weighted linear combination of state-action space features, to a nonlinear function obtained by training a neural network.

The idea sounds very much like a supervised learning problem. We can think of past observations as samples in our training set that we use to find a function that can map state observations to action values. However, in this case the training is done online, while the agent is exploring the environment. This raises the problem of samples being strongly correlated. Besides, while updating a particular element in the state-action table had no effect on the rest, changing a single parameter in our function can affect multiple states and actions.

3.1.1 Loss function

Finding a function that can effectively map state-action pairs to action values reduces to obtaining the set parameters that best approximate the action values in all regions. However, since the end goal is to find the optimal policy we might want to be able to approximate some states and actions better than others. Thus we need to decide on a suitable loss function than can help us asses performance. Like in most regression problems the obvious choice is the Mean Squared Error MSE, which can be weighted according to how much importance is given to each particular state-action pair:

$$L(\mathbf{w}) = \sum_{s \in S, a \in A} \mu(s,a)[q_\pi^*(s,a) - \hat{q}_\mathbf{w}(s,a)]^2, \tag{3.1}$$

with $\mu(s,a)$ proportional to the expected number of visits to a particular state and action compared to the rest. We can estimate the time spent at each pair as

$$\eta(s,a) = h(s) + \sum_{s'} \eta(s',a)\pi(a|s')p(s|s',a), \tag{3.2}$$

where h(s) is the probability that s is the initial state, and thus

$$\mu(s) = \frac{\eta(s)}{\sum_s \eta(s)}. \tag{3.3}$$

Hence if we knew the true action values for the optimal policy, the optimal set of parameters could be easily obtained by solving an optimization problem using equation 3.1 as objective function. However in a real scenario, since there is no prior knowledge of q_π^*, an approximation must be used instead.

3.1.2 Stochastic gradient and semi-gradient descent

We have already discussed various ways of approximating the value function. We could, for instance, substitute q_π^* by the MC estimate G_t in equation 3.1, and then, using stochastic gradient descent (SGD), update the parameters of our value function. Here we assume that $\mu(s, a)$ is the same for all state-action pairs, so that we have

$$\begin{aligned} \mathbf{w_{t+1}} &= \mathbf{w_t} - \frac{1}{2}\alpha\nabla \left[G_t - \hat{q}_{\mathbf{w}_t}(S_t, A_t)\right]^2 \\ &= \mathbf{w_t} + \alpha \left[G_t - \hat{q}_{\mathbf{w}_t}(S_t, A_t)\right] \nabla \hat{q}_{\mathbf{w}_t}(S_t, A_t), \end{aligned} \tag{3.4}$$

where α is the step size and $\nabla \hat{q}_{\mathbf{w}_t}(S_t, A_t)$ represents the gradient of $\hat{q}$ with respect to $\mathbf{w_t}$

Conversely, if we wanted to apply any of the TD methods described in section 2.6 we would have to use what is known in the literature as semi-gradient descent. The reason for this is that now, although our target also depends on the value function, it is not considered when calculating the gradient:

$$U_t = R_{t+1} + \hat{q}_{\mathbf{w}_t}(S_{t+1}, A_{t+1}) \tag{3.5}$$

$$\mathbf{w_{t+1}} = \mathbf{w_t} + \alpha \left[U_t - \hat{q}_{\mathbf{w}_t}(S_t, A_t)\right] \nabla \hat{q}_{\mathbf{w}_t}(S_t, A_t) \tag{3.6}$$

Although possibly compromising convergence, the use of equation 3.6 is justified in practice because it can notably speed up the learning process [3].

3.2 Deep Q Network

The algorithm presented in this section is in essence the same Q-learning method that helped us solve the deterministic grid world example except that now we use a neural network instead of a table to approximate the value function [4].

Randomly Initialize policy: π
Initialize value function parameters: $\mathbf{w}$
repeat
  Sample initial state: s
  while *not done* **do**
    $a_{max} \leftarrow \max_a \hat{q}_{\mathbf{w}}(s, a)$
    $a \leftarrow$ choose a_{max} or random a with probability ϵ and $1 - \epsilon$
    $s', r, done \leftarrow$ take step
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[r + \gamma \max_{a'} \hat{q}_{\mathbf{w}}(s', a') - \hat{q}_{\mathbf{w}}(s, a) \right] \nabla \hat{q}_{\mathbf{w}}(s, a)$
    $s \leftarrow s'$
  end
end

Algorithm 7: Off-policy TD method for control: Qlearning

The pseudo-code shown in algorithm 7 corresponds to the standard Q-learning algorithm with function approximation. A more complex version that incorporates a few additional elements, described below, was implemented in Python using Tensorflow. The algorithm and rest of the source code are available on GitHub[1]. The tutorial on RL with Tensorflow [16] was a valuable reference for the implementation of this method.

3.2.1 Experience Replay

As we mentioned in chapter 2, one important difference between reinforcement learning and supervised learning is that the samples we collect are not independent. Updating the network parameters in every iteration as we obtain new experiences can very much affect the stability of our optimization method.

Experience replay [17] intends to decouple information and break up correlations between successive samples by first storing the information in a big replay memory buffer and then sampling at random mini-batches of experiences in every iteration to effectively update the model. This has the effect of not only reducing variance and preventing instability issues, but also accelerating the learning process because experiences can be used more than once.

[1] https://github.com/miguelsuau/modified-ml-agents

3.2.2 Independent target network

We saw that when using the TD estimate as target to update our parameters a full gradient step could not be taken because our target was also dependent on the same parametric function. Although we saved this by using the so called semi gradient (equation 3.6), the fact that the function being updated is also present in our target means that they are both strongly correlated, which can provoke dangerous oscillations and make the algorithm diverge. The solution proposed in [4] consists of separating the two functions, the main network and the target network, and clipping the parameters of the target network. While the main network is updated in every iteration the weights of the target network are only periodically moved toward the main network after a certain number of iterations.

This two ideas, experience replay and the use of an independent target network, helped DeepMind build a model that achieved, and in some cases excelled, human level performance in a series of Atari video games [4].

3.2.3 Double Q-learning

Double Q-learning [18] aims to reduce the positive bias caused by the maximum operator used in the Q-learning update formula (equation 2.17) for both action selection and action evaluation. This effect, also known as overoptimism, has been shown to affect convergence, specially if suboptimal policies are repeatedly being favored.

The original paper [19], which is focused on tabular Q-learning, proposes storing two different tables, A and B, that learn the value function independently. The update is then performed using one of the functions to select an action and the other to estimate the value, so that

$$Q^A(S_t, A_t) = Q^A(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q^B \left(S_{t+1}, \arg\max_a Q^A(S_{t+1}, a) \right) \right] \tag{3.7}$$

and analogously for table B.

In every iteration, by random selection, only one of the two tables is updated so that they learn from different sets of experiences. The underlaying idea is to decouple the estimate from the action selection to remove the positive bias.

The authors successfully applied this same method to DQN but used the target network instead of a secondary function. This resulted in the following equation for calculating the targets:

$$U_t = R_{t+1} + \gamma \hat{q}\left(S_{t+1}, \arg\max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t), \mathbf{w}'_t\right), \tag{3.8}$$

where $\mathbf{w}_t$ and $\mathbf{w'}_t$ are the weights of the target and the main function respectively.

Although the two functions are now dependent, the use of the target network is justified because it only implies a minimal change in the algorithm. Besides, the fact that target network is only updated every once in a while has been proven enough to reduce bias and increase performance.

3.2.4 Static Grid World with function approximation

We showed that the tabular Q learning method could solve the static grid world example in very few iterations because the number of states-action pairs is small and the use of a parametric function is not necessary. Yet, we would like to test how well DQN does in the same task. Even though in [4] they use images as input and a convolutional neural network to encode observations, we will keep it simple and use a one-hot encoded vector representing the UFO's position at every time step.

The function approximator consists of a single fully connected layer with four nodes, each one corresponding to one of the four actions. This approach is only valid in situations where the number of actions available is small. In continuous control problems, for example, actions can be given as input to the neural network and then the function is maximized to find the greedy action. Another option is to treat the action space as Gaussian so that the function outputs mean and standard deviation for every state. The action is then sampled from the resulting distribution.

The algorithm stores experiences in a replay buffer and samples from it to update the function parameters. To make the optimization process more robust we added a secondary network to calculate the targets. Its weights are updated by a small amount toward the main network in every iteration. This has the same effect as a full update every certain number of steps but it prevents potential instability issues provoked by abrupt changes in the targets [16]. Finally, we

also applied double Q-learning, as described in the previous section, to reduce overoptimism.

The results of running DQN on the grid world example are shown in figure 3.1

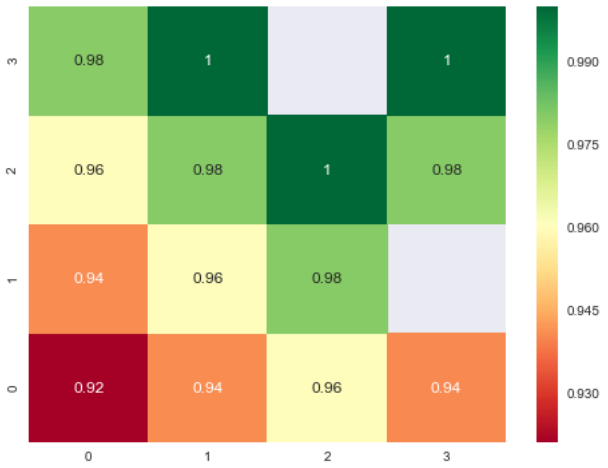


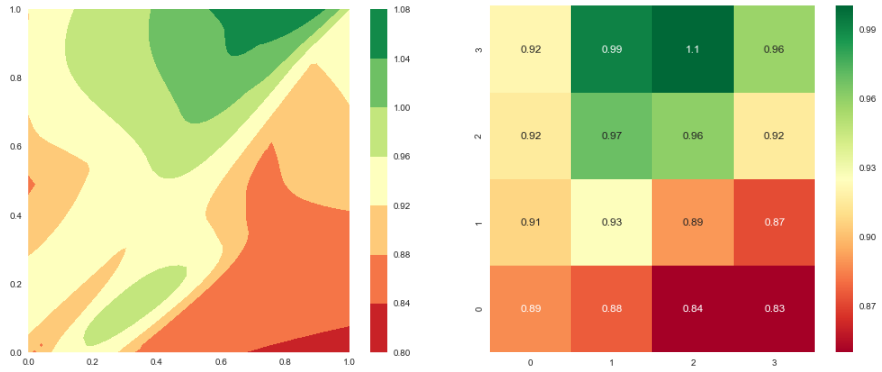
Figure 3.1: Value function of the static grid world example obtained using DQN. The values are equal to those obtained with tabular Q-learning

As we can see, using this binary state space representation, the method is able to match the values to the ones obtained with the tabular version. In fact, as long as the number of parameters is larger than the number of possible states-action pairs the neural network should be able to act like a regular table. In our case, because we use exactly 64 parameters and no bias terms, the weight values are equal to the 64 elements in the lookup table.

The above mentioned should be no surprise considering that our neural network is nothing but a linear regression between the binary vector as input, and the state-action values as output.

We could also feed our neural network with a more compact state representation. For this example we used a state space vector of two dimensions that correspond to the coordinates of the agent at each position. Figure 3.2 shows a contour plot of the value function in a continuous state space. Even though our agent is only allowed to occupy certain locations on the grid, the model can calculate the action value at any point.

Now the function is not flexible enough to reproduce the optimal value function



(a) **(b)**

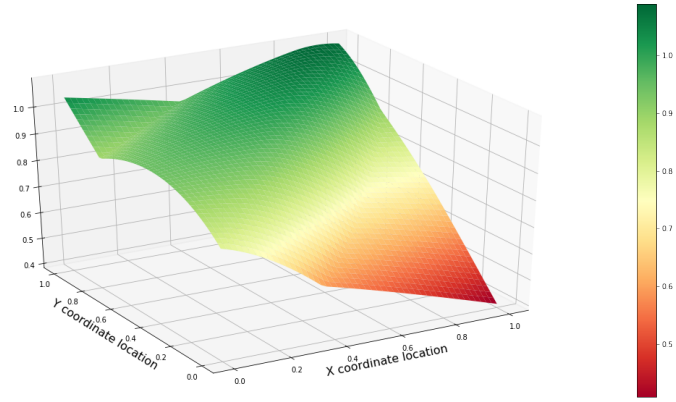
Figure 3.2: State-value function estimation using DQN. The colors in the contour plot (figure a) correspond to the estimated return of the greedy action at every possible location in the continuous state space. The values estimated by the neural network when feeding the allowed locations only are depicted in figure (b).

and needs to find a compromise solution. Yet, one can see that, although the values are not the true ones, the policy obtained when acting greedily is indeed the optimal policy. This example supports the use of a function approximator in complex reinforcement learning settings. As long as the function is able to describe the important elements in the environment it can very well be used to obtain the optimal policy. Figure 3.3 shows how the value function in 3D evolves during training, x and y axis are the coordinates in the grid while z gives the value at each location.

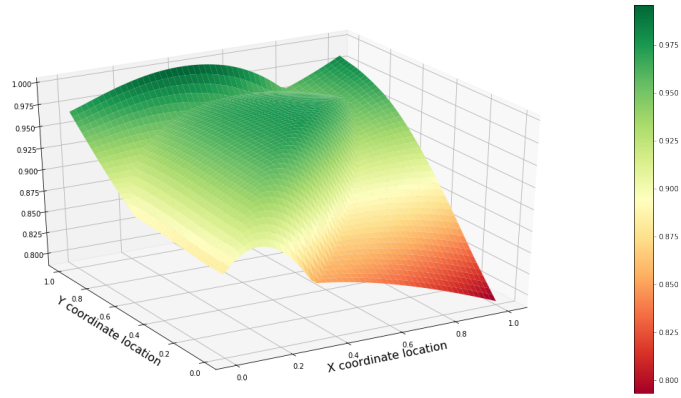
3.3 Policy Gradient Methods

All methods described so far were focused on learning a value function that was later used for selecting actions at every state. Considering that we are essentially interested in finding the best strategy that will allow the agent to act optimally, this was an oblique way of solving the control problem.

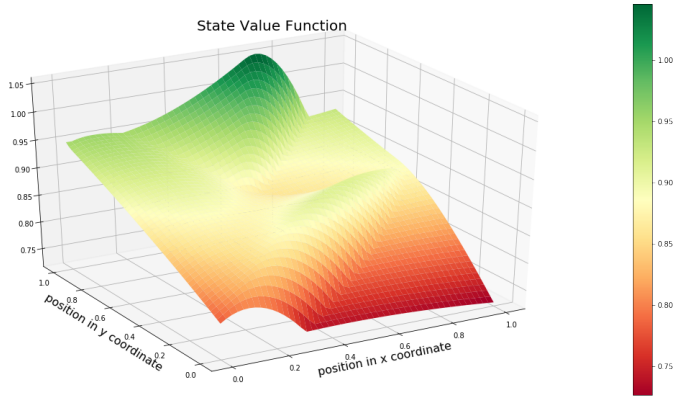
We present in this section a completely different approach in which we define a parametric function that can translate state observations into action probabilities [20]. This function, which is in fact the agent's policy, can be written



(a) 5e4 steps



(b) 2e5 steps



(c) 5e5 steps

Figure 3.3: State-value function estimation using DQN. The values shown in the z axis of the 3D graph correspond to the estimated return of the greedy action at every possible location in the continuous state space.

as

$$\pi(a|s,\theta) = Pr\{A_t = a|S_t = s, \boldsymbol{\theta_t} = \theta\} \tag{3.9}$$

We saw in the static grid world example that policies tend to be simpler than value functions and thus, easier to learn in practice. While before the value of an action had to increase over the rest in a given state in order to be selected, now any slight change to our parameters modifies the distribution over the actions, therefore affecting directly to our policy. Besides, policies are now non-deterministic, in the sense that unless the probabilities become exactly 0 actions have always some chance of being selected, this can help solve certain tasks where the optimal policy is a stochastic policy (e.g. the game of poker).

3.3.1 Policy objective function and gradient

As we did when we presented the function approximation methods, we need to define a loss function that will help us evaluate the performance of a particular policy:

$$\begin{aligned} J(\theta) &= E_\pi \left[\sum_{t=0}^{T} R_t \right] \\ &= \sum_a \pi_\theta(S_0, a) q_{\pi_\theta}(S_0, a). \end{aligned} \tag{3.10}$$

Equation 3.10 is nothing else than the expected return from the initial state S_0 when following policy π_θ from that point onwards. As mentioned earlier, because the policies are stochastic, updating our parametric function will slightly modify the chance of selecting certain actions but it rarely causes dramatic changes to our policy as it occurs with value based methods. This means that the objective function in 3.10 is smooth and thus, the method has better convergence properties [3].

The idea to find the optimal policy is to repeatedly adjust the parameters moving them in the direction that maximizes equation 3.10. A way to do so is by following the gradient, which according to the policy gradient theorem, whose

prove is given in [20], is calculated as follows

$$\begin{aligned}\nabla_\theta J(\theta) &\propto \sum_s \mu(s) \sum_a \nabla_\theta \pi_\theta(s,a) q_{\pi_\theta}(s,a) \\ &= E_{\pi_\theta}\left[\sum_a \pi_\theta(S_t,a) \frac{\nabla_\theta \pi_\theta(S_t,a)}{\pi_\theta(S_t,a)} q_{\pi_\theta}(S_t,a)\right] \\ &= E_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(S_t,A_t) q_{\pi_\theta}(S_t,A_t)\right]\end{aligned} \tag{3.11}$$

where $\mu(s)$ is the on-policy distribution over the state space under policy π as defined in equation 3.3.

The term $\nabla_\theta \log \pi_\theta(s,a)$ is often called eligibility vector and it indicates the direction along which a change in the network parameters will most increase the chance of selecting a particular action. On the other hand, $q_{\pi_\theta}(s,a)$, gives the expected return of a particular action given a state. Therefore, taking a step in the gradient direction will increase the probability of selecting actions that yield a high positive return while decreasing the probability of the ones whose return is negative.

3.3.2 MC Policy Gradient: REINFORCE

The most simple version of policy gradient methods is called REINFORCE [21] [22] and it uses an MC estimate of the action value to adjust the policy weights by applying stochastic gradient ascent:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \log \pi_\theta(S_t,A_t) G_t, \tag{3.12}$$

where $q_{\pi_\theta}(s,a)$ in equation 3.11 has been substituted by the sample return G_t. We first collect the experiences of an entire episode, and then update the parameters at every state using the sum of rewards from the current state until the end of the episode as an unbiased sample of the expected return. Algorithm

8 contains the pseudo-code describing the method.

Initialize policy parameters: θ
Allocate memory: states, actions, rewards
repeat
  Sample initial state: s
  G = 0
  while *not done* **do**
    a $\leftarrow$ choose an action at random from $\pi_\theta(s, a)$
    s', r, done $\leftarrow$ take step
    states, actions, rewards $\leftarrow$ append s, a, r
    $s \leftarrow s'$
  end
  for *each state-action pair, from terminal to initial state* **do**
    $G \leftarrow \gamma G + r$
    $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a, s) G$
  end
end

Algorithm 8: MC policy gradient: REINFORCE

3.3.3 TD Policy Gradient: Actor-Critic

As in any MC method, REINFORCE only updates the parameters of the policy once the episode is over which can slow down a lot the learning process. Besides, replacing the expected return with a sample estimate introduces high variance. One-step actor-critic methods [20] use a value function approximator as an estimate of the return after a single step. Hence, equation 3.12 becomes

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \log \pi_\theta(S_t, A_t) \hat{q}_{\mathbf{w}}(S_t, A_t). \tag{3.13}$$

The approach to obtain the action values is essentially the same as in value based methods with the only difference that now we only need the function to evaluate the actions and not to make decisions. Hence, we can distinguish between the actor π_θ, which defines our policy and therefore is the one influencing the agent's behavior, and the critic q_w, whose only task is to approximate the expected return.

The pseudo-code in algorithm 9 describes step by step how the one-step actor-critic method works. More sophisticated algorithms combine the properties of TD learning and MC methods using eligibility traces to trade off bias and

variance.

Randomly Initialize policy parameters: θ
Initialize value function parameters: $\mathbf{w}$
repeat
  Sample initial state: s
  $a \leftarrow$ sample an action at random from $\pi_\theta(s, a)$
  while *not done* **do**
    $s', r, done \leftarrow$ take step using a
    $a' \leftarrow$ sample an action at random from $\pi_\theta(s', a')$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[r + \gamma \hat{q}_{\mathbf{w}}(s', a') - \hat{q}_{\mathbf{w}}(s, a) \right] \nabla \hat{q}_{\mathbf{w}}(s, a)$
    $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a, s) \hat{q}_{\mathbf{w}}(s, a)$
    $s \leftarrow s'$, $a \leftarrow a'$
  end
end

Algorithm 9: TD policy gradient: one-step actor-critic

Although using a function approximator introduces some bias, it can also make the algorithm converge faster. Moreover, since the updates are done online the method is suitable for continuing control tasks.

3.3.4 Policy Gradient with Baseline

In order to reduce even more the variance of our estimates we define a baseline function [22], $B(S)$, and we subtract its value from the policy gradient. This operation must have no effect on the direction of the gradient and should only change its scale:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a \nabla_\theta \pi_\theta(s, a)(q_{\pi_\theta}(s, a) - b(s)) \tag{3.14}$$

and as long as the function does not vary with a

$$\sum_a b(s) \nabla_\theta \pi_\theta(s, a) = b(s) \nabla_\theta \sum_a \pi_\theta(s, a) = b(s) \nabla_\theta 1 = 0. \tag{3.15}$$

Most methods use the state value function $V_\pi(s)$ for the baseline so that the

term $(q_{\pi_\theta}(s,a) - b(s))$ in equation 3.14 becomes

$$D_\pi(S_t, A_t) = Q_\pi(S_t, A_t) - V_\pi(S_t). \tag{3.16}$$

$D_\pi(S_t, A_t)$ is known as advantage function and it measures the deviation of an action value from the current state estimate. Although it is common in the reinforcement learning literature to use A_π to denote the advantage function, note that this terminology would cause a name collision between our action variable A and the new function.

Whereas before we were updating actions based on their value estimate and, unless the estimated return was negative, we would adjust the parameters more or less strongly along the positive direction, now we penalize actions that are worse than the state value estimate no matter whether the outcome is positive or negative. In other words, the idea of using a baseline is that, in states where all actions have associated a positive return, we would like to discriminate between good and bad actions and increase or decrease their probabilities of being selected accordingly. This pushes our all gradient values to be in a neighborhood of 0 which is preferable to having gradients of very different magnitude depending on which state we are in.

One could then use this approach to reduce the variance of an actor-critic method and consequently, accelerate its learning rate. We could, of course, introduce a new function to approximate V_π but a more reasonable choice would be to use the critic. For the latter, we would modify the value function approximator so that it computes the state-value instead of the action value and then estimate the advantage at each time step using an approximation of the TD error:

$$\delta_{\mathbf{w}} = R_{t+1} + \gamma \hat{v}_{\mathbf{w}}(S_{t+1}) - \hat{v}_{\mathbf{w}}(S_t). \tag{3.17}$$

We can show that this is acceptable by substituting $\hat{v}_{\mathbf{w}}$ with the the true value function V_π and then taking the expected value:

$$\begin{aligned} E_\pi[\delta_\pi | S_t, A_t] &= E[R_{t+1} + \gamma V_\pi(S_{t+1}) - V_\pi(S_t) | S_t, A_t] \\ &= Q_\pi(S_t, A_t) - V_\pi(S_t) \\ &= D_\pi(S_t, A_t). \end{aligned} \tag{3.18}$$

The objective function that we are trying to optimize is then

$$J(\theta) = E[\log \pi_\theta(S_t, A_t) D_\pi(S_t, A_t)] \tag{3.19}$$

Algorithm 10 shows a variant of the Actor-Critic method that introduces a baseline function.

Randomly Initialize policy parameters: θ
Initialize value function parameters: $\mathbf{w}$
repeat
  Sample initial state: s
  $a \leftarrow$ sample an action at random from $\pi_\theta(s, a)$
  while *not done* **do**
    $s', r, done \leftarrow$ take step using a
    $a' \leftarrow$ sample an action at random from $\pi_\theta(s', a')$
    $\delta \leftarrow r + \gamma \hat{v}_{\mathbf{w}}(s') - \hat{v}_{\mathbf{w}}(s)$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}_{\mathbf{w}}(s, a)$
    $\theta \leftarrow \theta + \alpha\nabla_\theta \log \pi_\theta(a, s)\delta$
    $s \leftarrow s'$, $a \leftarrow a'$
  end
end

Algorithm 10: TD policy gradient: one-step actor-critic with baseline

3.3.5 Proximal Policy Optimization

Adding a critic in place of the MC sample return and a baseline function to scale the size of the objective gradient, certainly helped reducing the variance of the policy gradient methods. But considering that one major complication in reinforcement learning is the lack of a training set, performing a single parameter update in every iteration after collecting one sample does not seem to be the best way to use the limited amount of data that is available.

Because Q learning is an off-policy method when we implemented our Deep Q-learning algorithm, we used a big replay buffer that we would sample from to adjust the weights of our value function. This is not the case of policy gradient methods where the policy being optimized is the same we use to make decisions. Nonetheless, we can overcome this problem with importance sampling. This technique, which we already presented in section 2.5.2, allows us to optimize our parameters employing experiences obtained under a different policy. Following this idea Proximal Policy Optimization (PPO) [23] modifies the loss function in

equation 3.19 as follows

$$J(\theta) = E\left[\frac{\pi_\theta(S_t, A_t)}{\pi_{\theta_{\text{old}}}(S_t, A_t)} D_\pi(S_t, A_t)\right], \tag{3.20}$$

where $\pi_{\theta_{\text{old}}}(S_t, A_t)$ are the weights of our policy before the update. Substituting the objective function in algorithm 10 by equation 3.20, and repeatedly update our parameters using mini-batches of our experience buffer will, in most cases, make our policy diverge. As long as the old and new policies are similar there should no problem in using this new loss function, but as they become more and more different the magnitude of the gradient increases and so does the step size of our updates. Taking excessively large steps could modify the policy dramatically leading the agent to regions of the space from where it could be very difficult to recover.

To solve this issue PPO applies a variant of a numerical optimization technique called trust region. This idea was first introduced in [24] where the authors added an inequality constraint to the optimization problem that prevents old and new policies from diverging, by limiting their relative entropy (KL divergence) [14]. It turns out that this arrangement complicated the algorithm substantially and made it incompatible with certain deep learning architectures. Equation 3.21 is the objective function akin to trust region proposed in [23].

$$J^{\text{CLIP}}(\theta) = E\left[\min(r_t(\theta) D_\pi(S_t, A_t), \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) D_\pi(S_t, A_t))\right], \tag{3.21}$$

with

$$r_t(\theta) = \frac{\pi_\theta(S_t, A_t)}{\pi_{\theta_{\text{old}}}(S_t, A_t)}. \tag{3.22}$$

The clip operator acts like a boundary to the policy ratio. When the policies are too far from each other, the formula replaces the value of r_t by either $1-\epsilon$ or $1+\epsilon$, depending on whether D_t is positive or negative. We can then compute the gradient of equation 3.21 and use it to update our parameters multiple times sampling from a big pool of experiences.

PPO is considered the state of the art method in RL. It has been successfully applied to a wide range of different problems. Due to its versatility PPO is the method chosen by Unity ML-Agents. Since it is an open-source project the

algorithm implementation and all the necessary functions that allow to train agents in the Unity engine are publicly available on GitHub[2].

3.3.6 Dynamic grid world 1: DQN and PPO

In this section we will test and compare the performance of Unity's PPO implementation with our own version of DQN on the grid world example. In order to make it more challenging, we modify the environment so that the position of the objects is randomly selected in every episode.

Just knowing its own location the agent would be incapable of solving the problem. That's why, we also include among its state observations the position of the asteroid and the gold nugget at every time step. Therefore, the state representation consists of a 6 element vector containing the coordinates x and y for each of the objects. It might seem that from the agent's perspective the information fully describes the environment and should be enough to quickly obtain the optimal policy. However, since the vector of observations is continuous, the function approximator not only needs to estimate action values or actions distributions for the few allowed coordinate locations in the grid, but also for the infinite number of positions in between. In other words, the algorithm has to come up with a smooth and flexible neural network that can sufficiently describe a continuous version of the environment.

The performance comparison between DQN and PPO is depicted in figure 3.4. The plot on the right shows how the two methods are able to converge to a near optimal solution in 5×10^5 steps. Although in the static version of the environment we only penalized the agent with a negative reward of -0.01 per time step. With this choice the agent quickly learned to reach the goal but not always following the shortest path. It was only after more than 3×10^6 that both algorithms found an optimal solution. In figure 2.2 we used a penalty of -0.05 instead. This makes the agent worry more about reaching the gold nugget in fewer movements. Note that this change only decreases the value of the returns after every episode but it does not affect the optimal policy. The left plot in figure 3.4 shows how as the number of steps increases the episodes get shorter and shorter.

Since the best strategy in this environment is completely deterministic, in order to reach the optimal policy, the network parameters in PPO need to be adjusted so that the probability of selecting any of the suboptimal actions is exactly 0, therefore making it difficult for the method to finally converge, blue curve figure

[2] https://github.com/Unity-Technologies/ml-agents

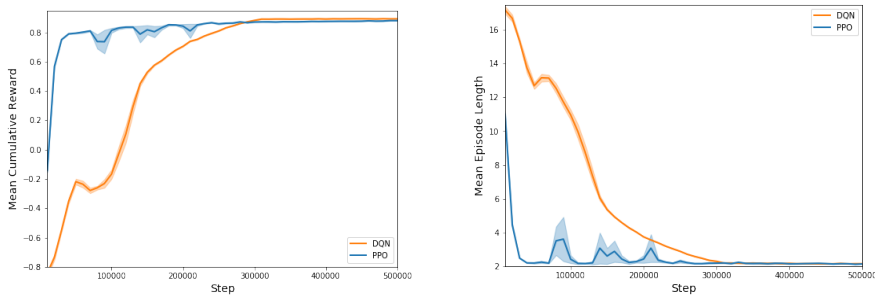


Figure 3.4: Comparison between DQN and PPO algorithms on the dynamic grid world environment. The thick lines are the mean cumulative reward and the mean episode length per time step averaged over 10 runs. The light blue and orange shaded areas represent the 1 standard deviation confidence interval of the mean values. For the same number of iterations DQN seems to reach a higher mean cumulative reward most likely because its policy is always deterministic. The turn back in the orange curve at 50K steps can be attributed to an abrupt change in the policy.

3.4. On the other hand, once the value estimates learned with DQN are close enough to the true values and the policy becomes greedy, the agent will always choose the action with the highest return and therefore act optimally, orange curve figure 3.4.

It is also worth mentioning that, although both algorithms take a similar number of steps to converge, PPO does so much faster in time. This is because in order for DQN to work well we need to keep a considerably large buffer of experiences which makes the code more demanding in terms of memory, and consequently slower. One solution that we adopted to speed up the process, was to maintain a fixed size for our array of samples, throwing away the oldest experiences as new ones came in. This worked well as long as the array was kept large enough, but the method failed to converge when making the buffer smaller.

It is not clear, that the oldest experiences are less important than the others. In fact, they are more susceptible to contain examples of bad action choices, which can actually be useful to approximate our value function. With this in mind we modified our code so that experiences are removed at random. However this seems to yield worse results in this particular example. A more sophisticated approach would be to smartly select which samples we want to remove. This is somewhat similar to the method called Prioritized Experience Replay proposed in [25]. The authors of this paper designed a framework where relevant experiences more likely to be sampled from the buffer when the model is updated.

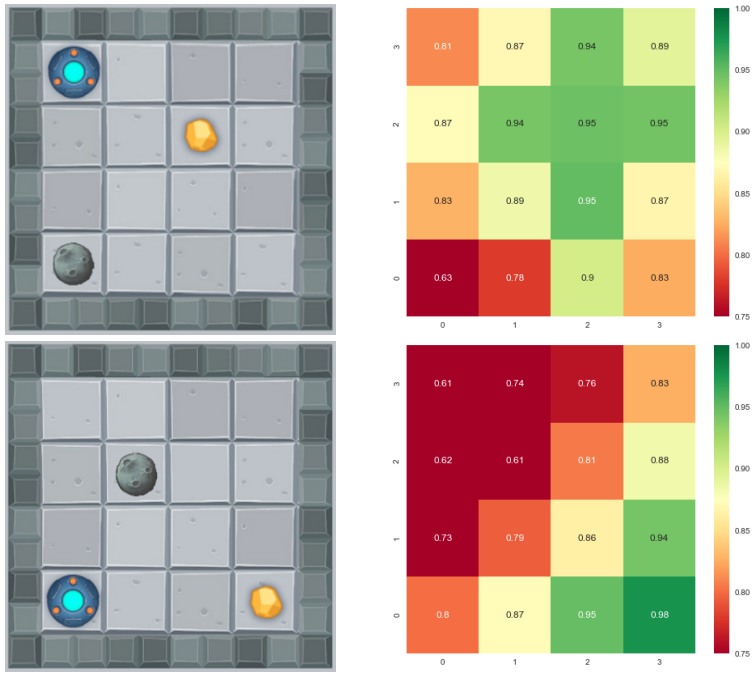


Figure 3.5: State-value function estimation using DQN. The values shown in the heat map on the right correspond to the state-values estimated by the neural network at each of the allowed locations when fixing the two objects position as shown in the screen shots. The values might differ from the true ones but they still lead to an optimal policy when acting greedily. Note that, in order to improve performance, the step penalty has been increased to -0.05, therefore altering the expected return at every state.

As explained before, because in DQN we select the actions that maximize the estimated return at every step, policies can experience abrupt changes even for small learning rates when one of the action values increases and exceeds the others, see turn back in orange curve at 50K steps in figure 3.4. In contrast, adjusting the parameters in policy gradient methods only causes slight changes in the distribution over actions. That is why the blue curve consistently moves towards the optimal values.

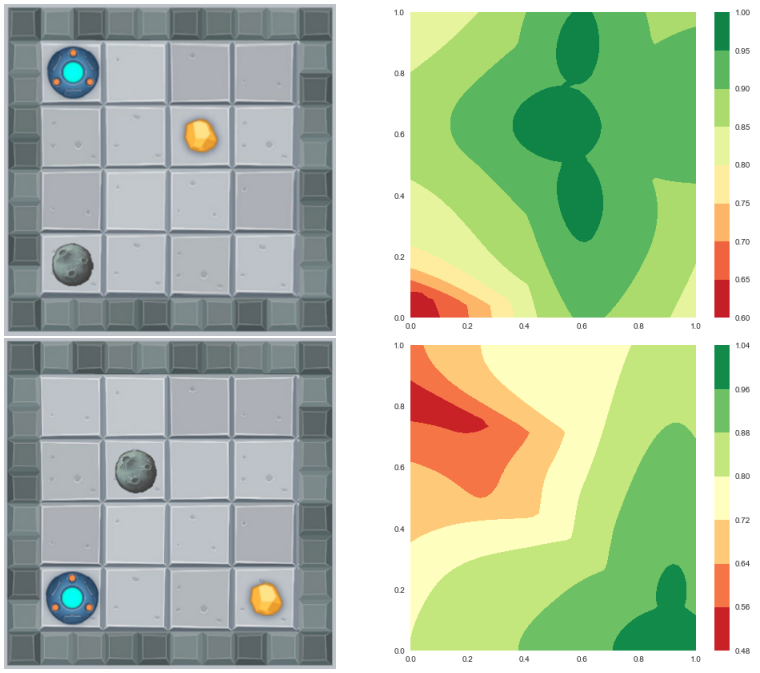


Figure 3.6: State-value function estimation using DQN. The values shown in the contours on the right correspond to the state-values estimated by the neural network at each location in the continuous state space when fixing the objects position as shown in the screen shots

In figure 3.5 we plotted the state-value function for different locations of the agent while maintaining the asteroid and the gold nugget fixed. Figure 3.7 on the other hand shows the stochastic policy obtained with PPO for two different settings. The size of the arrows in each of the cells reveals how likely is an action to be chosen, whereas the heatmap underneath shows the state value estimates.

Tables with the hyperparameters used for training and other details about this experiment are provided in the appendix. Follow the link [3] to find videos of the

[3]Dynamic Grid World 1: https://youtu.be/uQrNILVSy2I

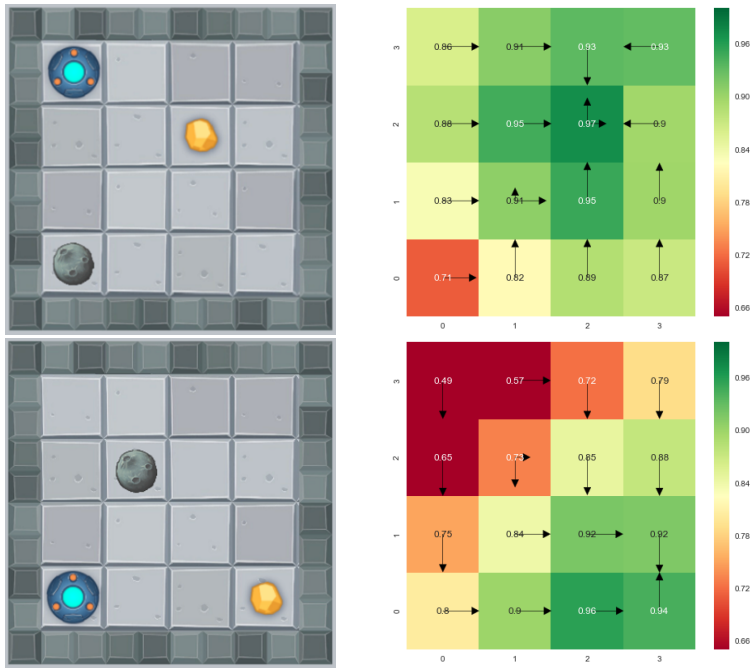


Figure 3.7: Policy and state-value function estimation using PPO. The size of the arrows in each of the cells reveals how likely is an action to be chosen. Only actions with probability above 10% are displayed. The heatmap underneath shows the state values estimates of the critic neural network at each of the allowed locations when fixing the two objects position as shown in the screen shots

results of this experiment.

3.3.7 Dynamic grid world 2: DQN and PPO

We now test the two methods on a more complicated environment. We increased the grid size an added an extra gold nugget and a second asteroid. The new configuration is depicted in figure 3.9. As in the previous example all the objects have different locations in every episode.

Apart from the increased complexity of having a larger grid and two extra objects, the agent needs to know whether or not a gold nugget has already been collected. To that end, we add an extra binary element to the vector of observations. This new element takes the value of 1 if the object remains in the grid and 0 if the agent already picked it up.

We again compare the performance of DQN and PPO in this new setting, figure 3.8. The turn back in the blue curve at 100K steps, probably caused by a sudden change in the policy, occurs when the agent learns to avoid the asteroids, episodes also become longer on average because the agent is too afraid to crash into them. In contrast, the orange curve, although showing some initial instabilities, which might be smoothen out with a proper parameter search, seems to learn more consistently and converge faster to the optimal policy.

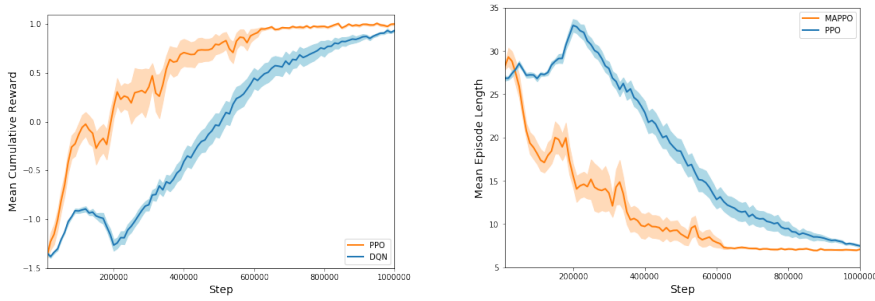


Figure 3.8: Comparison between DQN and PPO algorithms on the dynamic grid world environment. The thick lines are the mean cumulative reward and the mean episode length per time step averaged over 10 runs. Note that the blue curve shows a turn back at 200K steps which can be attributed to abrupt changes in the policies

In figure 3.9, we show the policy and the value estimates learned by PPO after 1M iterations. The screen shots on the right refer to two different states in the same episode. In the top one, we see how policy and value function give

more importance to the gold nugget on the lower left corner. After this one is collected, the object is set inactive, the value estimates change, and the arrows direct the agent towards the second goal.

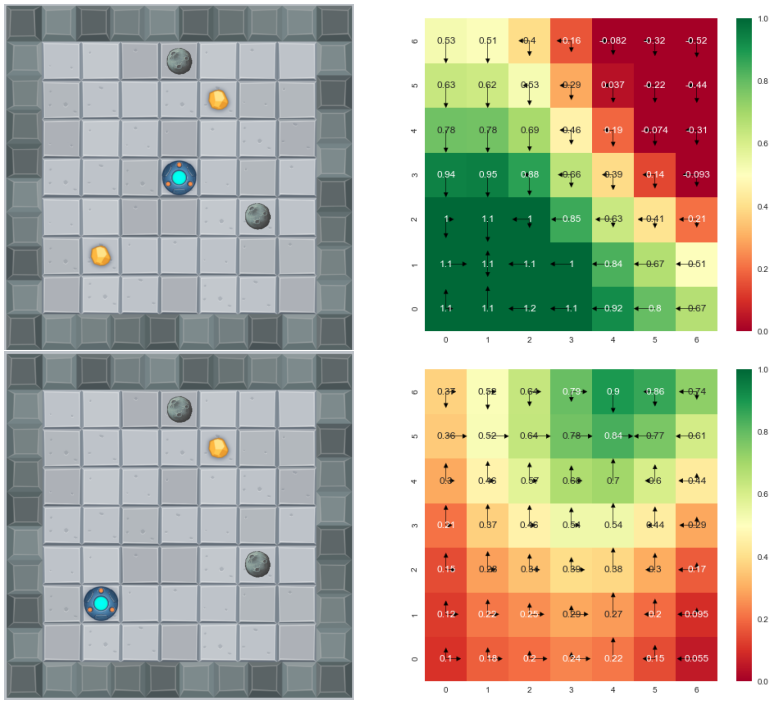


Figure 3.9: Policy and state-value function estimation using PPO. The size of the arrows in each of the cells reveals how likely is an action to be chosen. Only actions with probability above 10% are displayed. The heatmap underneath shows the state values estimates of the critic neural network at each of the allowed locations when fixing the objects position as shown in the screen shots. In the top heatmap the two golden nuggets are active, when the golden nugget located in $(1, 1)$ is collected (bottom screenshot) the object is set inactive, policy and value estimates change, and all the arrows direct the agent towards the second goal.

Tables with the hyperparameters used for training and other details about this experiment are provided in the appendix. Follow the link[4] to find a video showing the results of this experiment. Note that the policy is not optimal and the agent might not always select the shortest path to collect the two gold nuggets or might choose longer trajectories to stay away from the asteroid.

[4]Dynamic Grid World 2: https://youtu.be/vFZ34RTzAsM

Chapter 4

Multi-Agent Reinforcement Learning

In the second chapter we showed how the reinforcement learning framework can be applied to teach independent agents to effectively interact with the world and accomplish certain goals. We then presented a set of algorithms that using deep neural networks have been proven successful in solving very complicated problems. Nonetheless, the use of only one agent might not be sufficient in scenarios that require team collaboration. Although we are used to see examples of groups of robots and electronic devices working together in the same task, these are often programmed to follow certain heuristic rules and they would probably not perform well in other, more complicated, settings.

In this chapter we introduce the idea of Multi-Agent Reinforcement Learning. We start by explaining why the concept of MDP is no longer valid and why the algorithms we used so far could fail when trying to coordinate multiple agents. We then present the Markov Games model as an alternative to MDPs and define some useful concepts from Game Theory. Finally, we outline the main features of two new algorithms and provide some examples that reveal why these are more suitable for multi-agent systems.

4.1 Non-Stationary Markov Decision Process

Perhaps the main assumption we made when formulating the decision making problem as an MDP was that the global dynamics of the environment remained constant throughout the entire process and thus, each particular state and reward in an episode were only dependent on the immediately preceding state and action pair. In this context, the transition and reward functions were unalterable and the agent could only optimize its own policy in order to obtain the maximum reward.

This assumption is not well-founded when bringing other agents into play. The reason is that these new agents, who are also learning to interact with the environment, might alter the transitions from the individual agent's perspective, making the process non-stationary [26]. Besides, a change in the policy of any of the agents might have an effect on each agent's optimal policy making it difficult for the process to ever converge. Although in practice, traditional RL algorithms such as Q-learning or Policy Gradient have been shown to work well in certain multi-agent environments [9] their use is not justified from the theoretical perspective since the actions of other agents can cause instabilities in the learning dynamics of each individual agent [26].

4.2 Markov Games

The Markov Games (MG) framework, also known as Stochastic Games [27], can be used to adapt the MDP view point to a multi-agent system and is defined as the tuple $\langle S, U, O, \Omega, T, R \rangle$ where S and U are the sets of possible states and joint actions of all agents. Ω represents a mapping between the true states S and the set of joint observations O. T defines the transitions between different states and R is the reward function. Because the transition and reward functions associated to each agent are now dependent on all the agents' actions the process can be again considered stationary.

As in the single agent case, each of the agents learns its own policy by maximizing the discounted sum of rewards

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{4.1}$$

where γ is the discount rate and T the time horizon.

4.3 Selecting collective goals and rewards

It has already been mentioned that one of the biggest challenges in RL is choosing a reward function that complies with the overall objective. Partial rewards need to be selected so that the agent is unequivocally lead to solving the task. This issue is even more dramatic when the need for team coordination is required. In certain cases, a wrong selection might make the agents compete instead of work together.

Applying concepts from the field of game theory we define three possible scenarios depending on how partial rewards are selected.

A. Fully competitive or zero sum games: Agents have opposite reward functions. In the most common setting two agents that are opponents compete for a common goal. One agent's success is translated into failure for the other. Board games such as chess or backgammon can be treated as fully competitive tasks and solved using techniques like the minimax-Q algorithm [27]:

$$\pi_i^*(s) = \arg\max_a \min_{\mathbf{o}} Q^*(s, a, \mathbf{o}), \tag{4.2}$$

where a is the action of the one agent whose policy is evaluated and $\mathbf{o}$ are the actions of its opponents.

B. Fully cooperative games: The agents share the same reward function. All team members are equally penalized when any of them makes a bad choice but also rewarded when they act correctly. Consequently, each agent's optimal policy is the one that maximizes the centralized action value function

$$\pi_i^*(s) = \arg\max_{u_i} \max_{\mathbf{u} \in U} Q^*(s, \mathbf{u}), \tag{4.3}$$

where $\mathbf{u}$ represents the actions of all agents and u_i the action choice of agent i.

C. Mixed or general sum games: Reward functions are neither equal nor contrary. In this setting individual agents might make decisions that are in conflict with the overall objective. Agents, acting according to their own advantage, could compromise the outcome of the group task if that brings them greater reward. Although this type of game is very common in reality, finding an equilibrium point is particularly difficult in practice. In Game Theory, a situation where none of the players can benefit from

changing only their own strategy while the other players' strategies remain unchanged is known as Nash equilibrium and constitutes a solution to the general sum games. A more formal definition of the Nash equilibrium can be found in [28]

All the examples that we present in this thesis can be classified either as fully cooperative or mixed games.

It is not always obvious when designing an environment how the agents will react to the different partial rewards that are imposed. In our case, the examples had to be modified and trialled several times before the desired behavior was obtained.

4.4 Centralized and decentralized algorithms

We now introduce different techniques that present an alternative to applying traditional RL methods to environments with more than one intelligent agent, which do not follow the Markov property.

One could address the non-stationarity issues by introducing a global entity that has access to all the agents observations and controls all the agents actions in a completely centralized manner. The problem will be then equivalent to a single agent control problem. Yet, this option, aside from not presenting any extra challenge, could not be available in certain cases where the agents might be required to act independently.

Following the stochastic games framework, agents can also learn independent policies, which take as input both the current observation and the actions other agents might take, and return the action that maximizes the centralized action value function. To accomplish this, individual agents need to keep track of any changes in the other agent's policies.

This is the idea followed by the Nash Q-learning algorithm [29] where the agents need to find the optimal strategy according to their own model of the other agents Q functions. In other words, in every step the agents make decisions about what actions to take based on what they believe the other agents will do.

Nash Q convergence guarantees are limited to a reduced number of cases where certain conditions need to be satisfied [30]. A more general method with better convergence properties, called Friend or Foe Q-learning [31], maintains a single

Q function which is updated using either an augmented version of the ordinary Q learning, which maximizes over the joint action space, or the minimax equation used for solving zero-sum Markov games, depending on whether the opponents are considered friends or foes.

Decentralized Partially Observable MDP (Dec-POMDP) [32] is a popular model that assumes transitions and rewards satisfy the markovian property and depend only on the preceding true state and joint actions. However, each individual agent's policy has no access to this information. Similarly, agents can only base their decisions on their own observations and not on the true states or the other agent's observations.

In the following sections we introduce two algorithms that despite requiring other agent's information to update or evaluate their policies during training, the learned functions are completely decentralized and do not precise of any knowledge of the other agents intentions during execution.

4.5 Multi-Agent DQN: Alternating Maximization

The algorithm presented here is based on a method normally referred as alternating maximization, this concept was first introduced in [33] and then adapted to model free deep RL in [34]. The idea is to train one of the agents while freezing the other agents' policies. From the objective agent's perspective the other agents are part of the dynamics of the environment and have no influence on the transition and reward distributions. After a certain number of iterations the updated policy is shared with the other agents and the sequence starts again until convergence.

Although the overall process is still non-stationary, the training happens in multiple phases within which the transitions and rewards remain unchanged.

The pseudo-code of a simple multi-agent Q-learning method is outlined in algorithm 7. Even though the network parameters of agent 1 are shared during training, the policies learned are completely decentralized because the value

function is only dependent in the current state and partial action of each agent.

Randomly Initialize policy parameters: $\theta_1, \theta_2, ..., \theta_n$
Initialize value function parameters: $\mathbf{w}_1, \mathbf{w}_2, ..., \mathbf{w}_n$
repeat
  repeat
    Initialize: s
    while *not done* **do**
      $a_{1\,\max}, a_{2\,\max}, ..., a_{n\,\max} \leftarrow$
       $\max_{a_1, a_2, ..., a_3} \{\hat{q}_{\mathbf{w}_1}(s, a), \hat{q}_{\mathbf{w}_2}(s, a)..., \hat{q}_{\mathbf{w}_n}(s, a)\}$
      $a_1, a_2, ..., a_n \leftarrow$ choose $a_{1\,\max}, a_{2\,\max}, ..., a_{n\,\max}$ or random
       $a_1, a_2, ...a_n$ with probability ϵ and $1 - \epsilon$
      $s', r, done \leftarrow$ take step
      $\mathbf{w}_1 \leftarrow \mathbf{w}_1 + \alpha \left[r + \gamma \max_{a'_1} \hat{q}_{\mathbf{w}_1}(s', a'_1) - \hat{q}_{\mathbf{w}_1}(s, a_1) \right] \nabla \hat{q}_{\mathbf{w}_1}(s, a_1)$
      $s \leftarrow s'$
    end
    Copy parameters: $\mathbf{w}_2, ..., \mathbf{w}_n \leftarrow \mathbf{w}_1$
  end
end

Algorithm 11: Multi-Agent Q-learning: alternating maximization

Our implementation applies the same DQN algorithm described in the previous chapter to update the policy, with the only difference that the experience replay buffer is emptied every time the other agent's policies are updated. This ensures that experiences corresponding to previous policies are not used for the updates.

The method assumes that the role each of the agents plays in the environment is the same. Nonetheless, it can be easily adapted to a more general setting by taking turns training the agents instead of copying the parameters from one network to another.

The algorithm and rest of the source code are available on GitHub [1].

4.5.1 Multi-Agent grid world: Competition

In our first multi-agent example we make our agents compete for the gold nuggets. To do so, we modify our grid world environment so that it now includes an additional agent. The agents receive an individual reward of +1 when collecting the pick up objects and they are penalized with -1 when crashing into an asteroid. Note that the environment is not fully competitive, since the agents do not have opposite reward functions, and therefore, it can not be classified

[1] https://github.com/miguelsuau/modified-ml-agents

as a zero sum game. The screen-shots (a) and (b) in figure 4.1 show the two different settings that we use in this experiment.

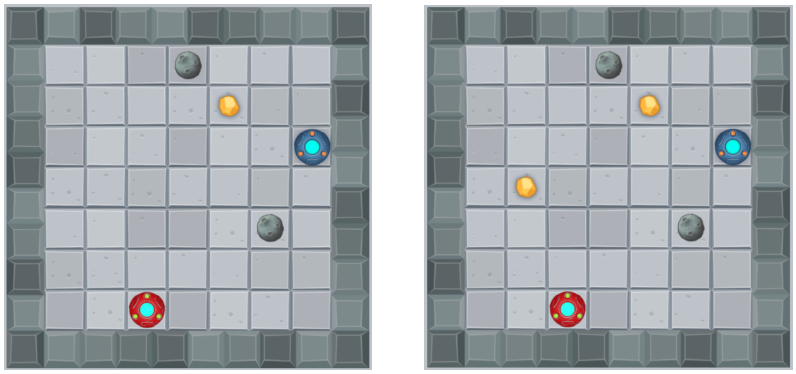


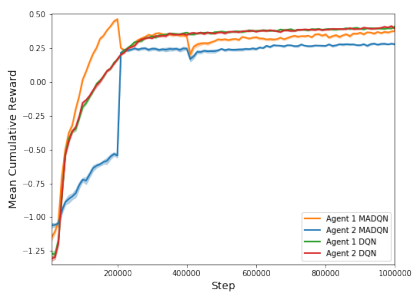
Figure 4.1: Multi-Agent grid world example with 2 agents , 2 asteroids and 1 (a) or 2 gold nuggets (b). The position of all the objects changes from one episode to another

The performance of the multi-agent DQN method is assessed by testing it against our original DQN implementation. Figure 4.2 shows how, although the presence of a second agent makes the environment non-stationary from the MDP perspective, agents trained with DQN are still able to find an optimal policy when competing for only one gold nugget. The method, however, has more difficulties when having two goals. In this second case, MADQN agents seem to reach higher returns and shorter episode lengths. It is important to mention that the multi-agent method has not converged after 1M steps. This can be easily seen in figures 4.2a and 4.2c, where there is still a gap at 1M steps between the mean cumulative reward of the two agents, meaning that they have not yet found the Nash equilibrium.

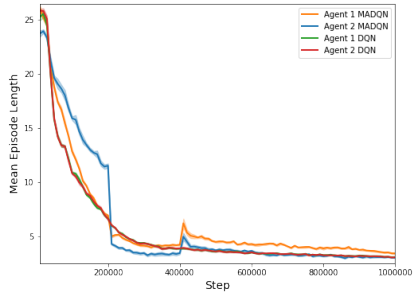
There is an extra parameter in the new Multi-Agent DQN algorithm that must be carefully selected. This is the total number of steps between two updates of the frozen policies. The parameter should be large enough so that the free agent has time to learn about the new environment dynamics and develop its policy.

Tables with the hyperparameters used for training and other details about this experiment are provided in the appendix. Follow the link[2] to find videos of the results of this experiment. Note that because the agents need to reach the gold nuggets before the other does, they are willing to take more risks and thus, they are more likely to crash into an asteroid.

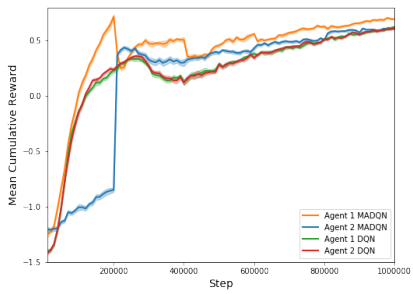
[2]Multi-Agent Competition 1: https://youtu.be/5rKOEfChKDM
Multi-Agent Competition 2: https://youtu.be/s-L2ECkeso0



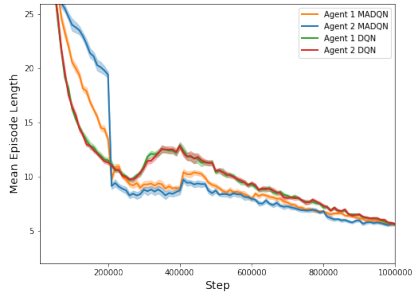
(a) 1 goal



(b) 1 goal



(c) 2 goals



(d) 2 goals

Figure 4.2: Competitive multi-Agent grid world. The thick lines are the mean cumulative reward (a, c) and the mean episode length (b, d) per time step averaged over 10 runs. The shaded areas represent the one standard deviation confidence interval of the mean values. We used a random seed and the same hyperparameters for both algorithms. Even though the presence of a second agent makes the MDP non-stationary ordinary DQN can still find a policy that leads to a higher mean cumulative reward than the one obtained when training the agents with MADQN in the single goal case. Contrarily, MADQN seems to outperform DQN when there are two gold nuggets in the environment.

4.6 Multi-Agent PPO: Centralized Critic

The following algorithm consists of a policy gradient method with a centralized critic [10]. The algorithm learns a state value function for each agent, which takes as input the current observation and the actions of all the other agents, and uses it to evaluate each agent's actions and update the policy network. Following the policy gradient theorem, equation 3.11 becomes

$$\nabla_{\theta_\mathbf{i}} J(\theta_\mathbf{i}) = E_{\pi_{\theta_i}} \left[\nabla_\theta \log \pi_{\theta_i}(S_t, A_t) V_i(S_t, \mathbf{U}_t) \right], \tag{4.4}$$

where $\mathbf{U}_t$ are the actions chosen by the other agents at time t within the sets of all possible actions $\{\mathcal{A}_2, \mathcal{A}_3, ..., \mathcal{A}_n\}$. As we can see only the critic depends on the joint set of actions while the individual policies rely exclusively on local information.

Like in DQN we can use the mean squared error as loss function to update the parameters of our neural network

$$\mathcal{L}(\mathbf{w_i}) = E[(V_i(S_t, \mathbf{U}_t) - y)^2], \tag{4.5}$$

with y being the one step TD estimate

$$y = R_{t+1} + \gamma V_i(S_{t+1}, \mathbf{U}_{t+1}). \tag{4.6}$$

According to equation 4.6, it is required that the local policies are shared during training so that each agent can obtain samples of U_{t+1} in every update.

Algorithm 12 is a version of the one step policy gradient method that uses a joint action value function as critic. In our case, we extended the critic network in Unity's PPO implementation to take the other agents' actions as input apart from the state information. In order to reduce bias in both the policy sample gradient and the critic loss function the algorithm applies an n-step TD estimate. Moreover, as in the original PPO algorithm it also incorporates a baseline

function to reduce variance.

Initialize policy parameters: $\theta_1, \theta_2, ..., \theta_n$
Initialize value function parameters: $\mathbf{w}_1, \mathbf{w}_2, ..., \mathbf{w}_n$
repeat
  Sample initial state: s
  $a_1, a_2, ..., a_3 \leftarrow$ sample actions at random from $\pi_{\theta_1}(s), \pi_{\theta_2}(s), ..., \pi_{\theta_n}(s)$
  while *not done* **do**
    $s', r, done \leftarrow$ take step using $a_1, a_2, ..., a_n$
    $a'_1, a'_2, ..., a'_3 \leftarrow$ sample actions at random from
     $\pi_{\theta_1}(s'), \pi_{\theta_2}(s'), ..., \pi_{\theta_n}(s')$
    for *agent i = 1 to n* **do**
      $\mathbf{u} = a_j$ where $j \in 1, 2...n$ and $j \neq i$
      $\mathbf{w}_i \leftarrow \mathbf{w}_i + \alpha \left[r + \gamma \hat{v}_{\mathbf{w}_i}(s', \mathbf{u}') - \hat{v}_{\mathbf{w}_i}(s, \mathbf{u})\right] \nabla \hat{v}_{\mathbf{w}_i}(s, \mathbf{u})$
      $\theta_i \leftarrow \theta_i + \alpha \nabla \log \pi_{\theta_i}(a, s) \hat{v}_{\mathbf{w}_i}(s, \mathbf{u})$
    end
    $s \leftarrow s'$
    $a_1, a_2, .., a_n \leftarrow a'_1, a'_2, ..., a'_n$
  end
end

Algorithm 12: MA Policy Gradient: Centralized Critic

Finally to make the code more efficient, instead of using two separate functions for actor and critic, we build a single network and append all the actions to the output of the last hidden layer. We have then a softmax layer for our policy, which only takes as input the state information, and a fully connected layer that returns the state values. The loss function used to optimize the network is therefore a combination of the policy loss and the value loss.

As opposed to [10] our method approximates the state-value function $V(S_t, \mathbf{U}_t)$ instead of the action-value function $Q(S_t, A_t, \mathbf{U}_t)$ so that we can then apply generalized advantage estimation GAE [35]. An exponentially weighted estimator that combines the MC sample return and the TD estimate to approximate the advantage function D_t:

$$\hat{D}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{4.7}$$

with

$$\delta_t = R_t + \gamma V(S_{t+1}) - V(S_t) \tag{4.8}$$

where $\lambda \in [0, 1]$ in equation 4.7 intends to measure how much a particular action is to blame for a specific long term reward. This is normally referred as credit assignment problem [3].

On the other hand, as in the original PPO [23] algorithm the policy is updated with mini-batches of experiences sampled from a replay buffer and adjusted to refer to the current policy applying importance sampling. This allows to perform multiple updates using past experiences and therefore make better use of the information.

The authors of [10] propose having a separate network to calculate the target which like in DQN is only updated every certain number of iterations. After testing this approach on two different environments we found out that it seems to have very little, if any, effect on our method.

The algorithm and rest of the source code are available on GitHub[3].

4.6.1 Multi-Agent grid world: Cooperation

We now modify the previous multi-agent environments and make them fully collaborative. To that end, we set the rewards to be equal for both agents. That is, the two agents are rewarded when a gold nugget is collected and penalized when any of them collides with an asteroid. As usual, they are also penalized in every step.

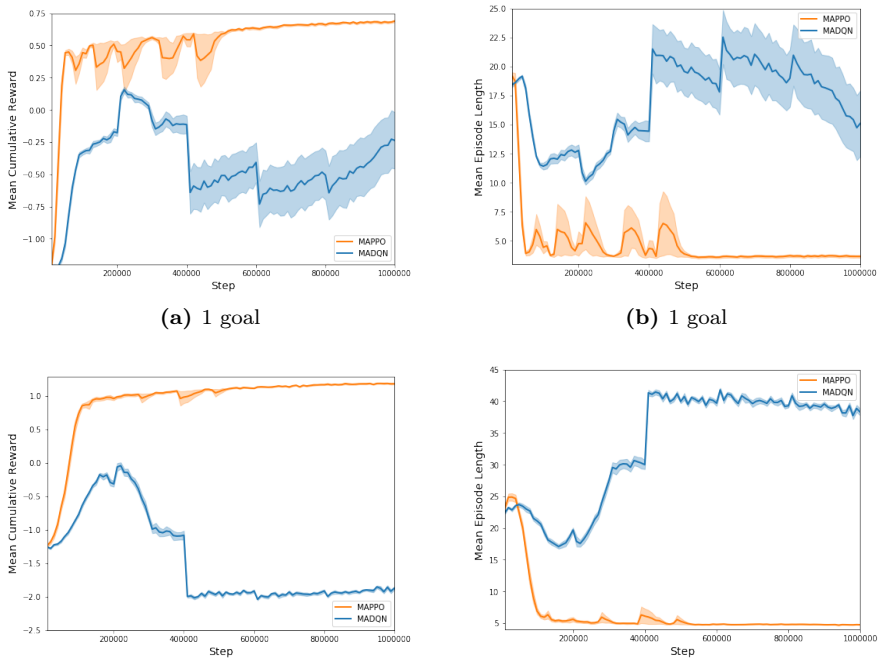
One might argue that blaming the two agents when one of them steps on an asteroid can only confuse the other agent and make the learning process more difficult. Nonetheless, if only the one making the bad choice receives a penalty, they might learn to not take any risks and let the other collect the gold nuggets, which will definitely affect collaboration.

The above mentioned is actually what happens when training the agents with MADQN, figure 4.5. The agent whose policy is being continuously optimized quickly learns to collect the goals, but when the value function parameters are copied at 200K steps, it realizes that it is safer to stay still and wait for the other agent to pick up the gold nuggets. This causes a dramatic decrease in the average episode returns at 400K steps, when the second agent acquires the new policy. The thick lines are the mean cumulative reward (a, c, e) and the mean episode length (b, d, f) per time step averaged over 10 runs. The light blue and orange shaded areas represent the one standard deviation confidence interval of

[3] https://github.com/miguelsuau/modified-ml-agents

the mean values. We used a random seed and the same hyperparameters for both algorithms.

In contrast, because the training occurs simultaneously in MAPPO, despite the initial oscillations, which are most likely provoked by the same issue, the agents recover and learn to work together and gather the pick up objects.

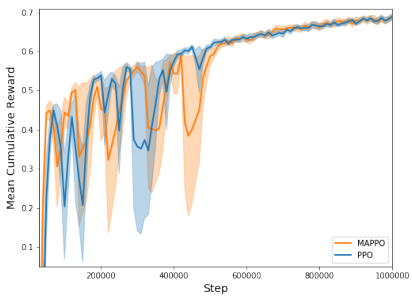


(a) 1 goal **(b)** 1 goal

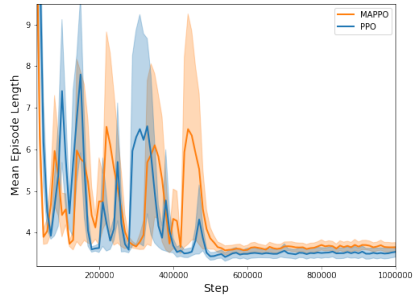
(c) 2 goals **(d)** 2 goals

Figure 4.3: Cooperative multi-agent grid world. Agents trained with MADQN for 1M steps are unable to work together. Even though the unfrozen agent initially learns to collect the pick ups, when the parameters are copied at 200K steps, it realizes that it is safer to wait for the other agent to do the work, which causes the large drop at 400K steps, just when the information is transfered again. On the other hand, despite the initial instabilities MAPPO agents can successfully coordinate and collaborate to reach the goals.

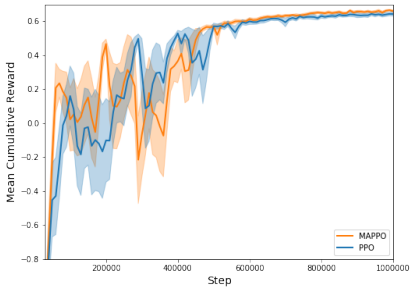
In this particular case, we include the option to stand still in the set of actions that are available to the agent. The idea is to see whether an agent can recognize that the other is in a more favorable position and therefore remain static.



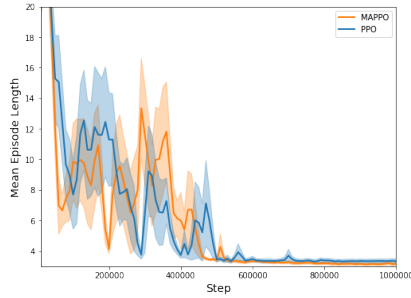
(a) T = 1



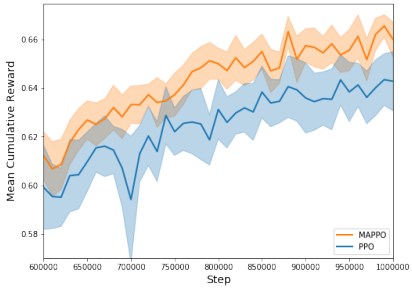
(b) T = 1



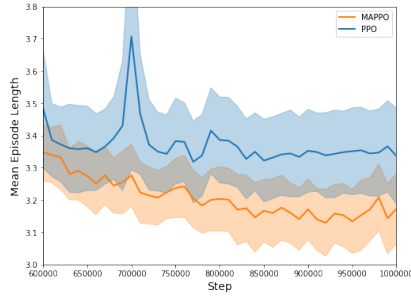
(c) T = 0



(d) T = 0

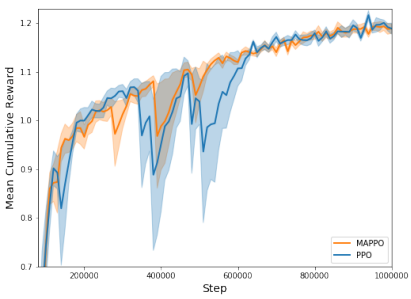


(e) T = 0, zoom

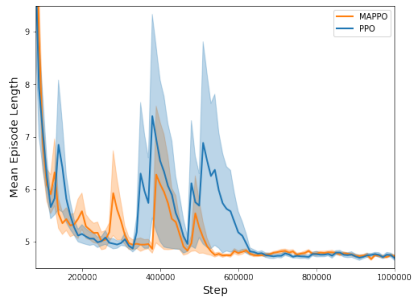


(f) T = 0, zoom

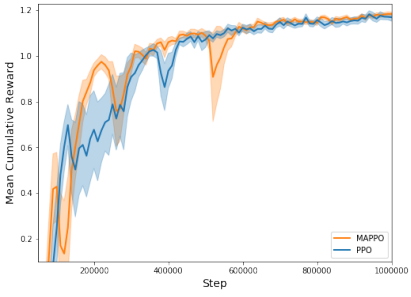
Figure 4.4: Cooperative multi-agent grid world. Agents trained with MADQN for 1M steps are unable to work together. Even though the unfrozen agent initially learns to collect the goals, when the parameters are copied at 200K steps, it realizes that it is safer to wait for the other agent to do the work, which causes the large drop at 400K steps, just when the information is transfered again. On the other hand, despite the initial instabilities MAPPO agents can successfully coordinate and collaborate to pick up the gold nuggets.



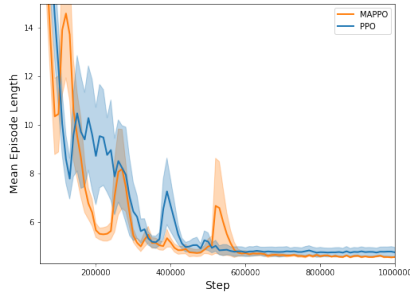
(a) T = 2



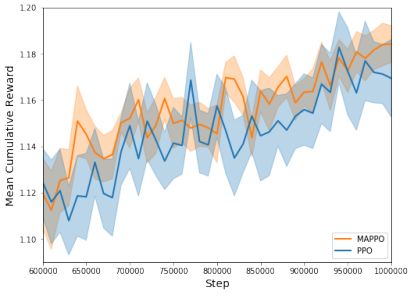
(b) T = 2



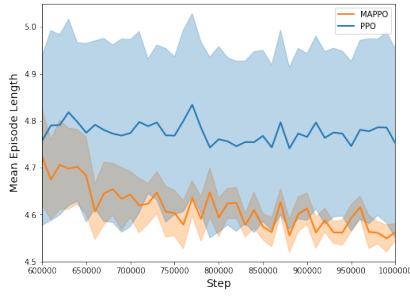
(c) T = 1



(d) T = 1



(e) T = 1, zoom



(f) T = 1, zoom

Figure 4.5: Cooperative multi-agent grid world. Agents trained with MADQN for 1M steps are unable to work together. Even though the unfrozen agent initially learns to collect the goals, when the parameters are copied at 200K steps, it realizes that it is safer to wait for the other agent to do the work, which causes the large drop at 400K steps, just when the information is transfered again. On the other hand, despite the initial instabilities MAPPO agents can successfully coordinate and collaborate to pick up the gold nuggets.

Tables with the hyperparameters used for training and other details about this experiment are provided in the appendix. Follow the link [4] to find videos showing the results of this experiment. Note that the agents sometimes wait for the other to collect the gold so as to avoid a potential collision. In fact, in the first video with only one gold nugget, we can see how the red agent tends to stay always at the initial position and only moves when it is clearly the closest to the goal.

4.6.2 Multi-Agent grid world: Learning to communicate

In this new scenario, the agents are provided with a communication channel which they can use to send messages to each other. This messages are nothing else than binary vectors which the agents can then convert into more meaningful concepts. This example is very similar to the one given in [10], although in this case we use our MAPPO implementation to train our agents.

The environment is depicted in figure 4.6. In each episode, the blue agent, which is color blind, must collect either the yellow, the green or the red pickup. The objective is indicated by the color of the frame which only the red agent situated in the southeast corner can see. The red UFO, who plays the role of speaker, can only send binary messages to the blue UFO, the listener. These messages become part of the state information of the blue agent along with the coordinates of the two pickup objects. Like in the other sample environments, the blue agent can take 4 actions moving north, south, east or west.

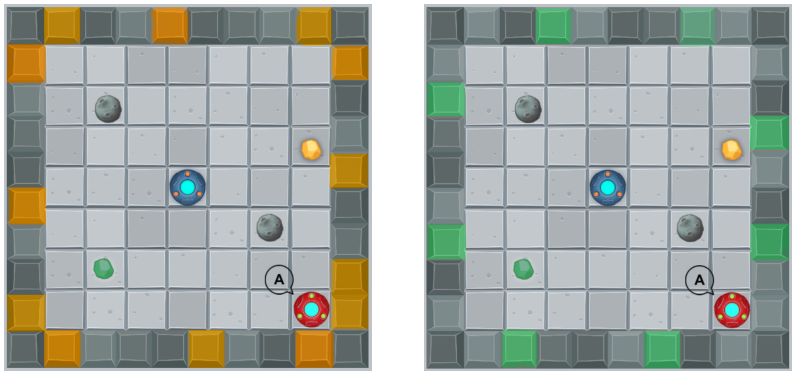
The complexity of this environment relies on the fact that the two agents need to understand each other and develop a language to communicate without any prior knowledge. This involves that the speaker needs to learn to be consistent on its messages so that the listener always receives the same state information for each of the objectives.

It is important to mention that the communication is not simultaneous which might lead in some occasions to a change in the blue UFO direction. The listener receives the message sent by the speaker after one time step.

Figure 4.7 is a comparison between Unity's PPO implementation and our multi-agent version. The curves are the result of averaging mean cumulative reward and mean episode length over 10 executions.

The top graphs, figures 4.7a and 4.7b, reveal that, contrary to what we ex-

[4] Multi-Agent Cooperation 1: https://youtu.be/v_c4fjqwt0g
Multi-Agent Cooperation 2: https://youtu.be/mJO2z8c_UiE



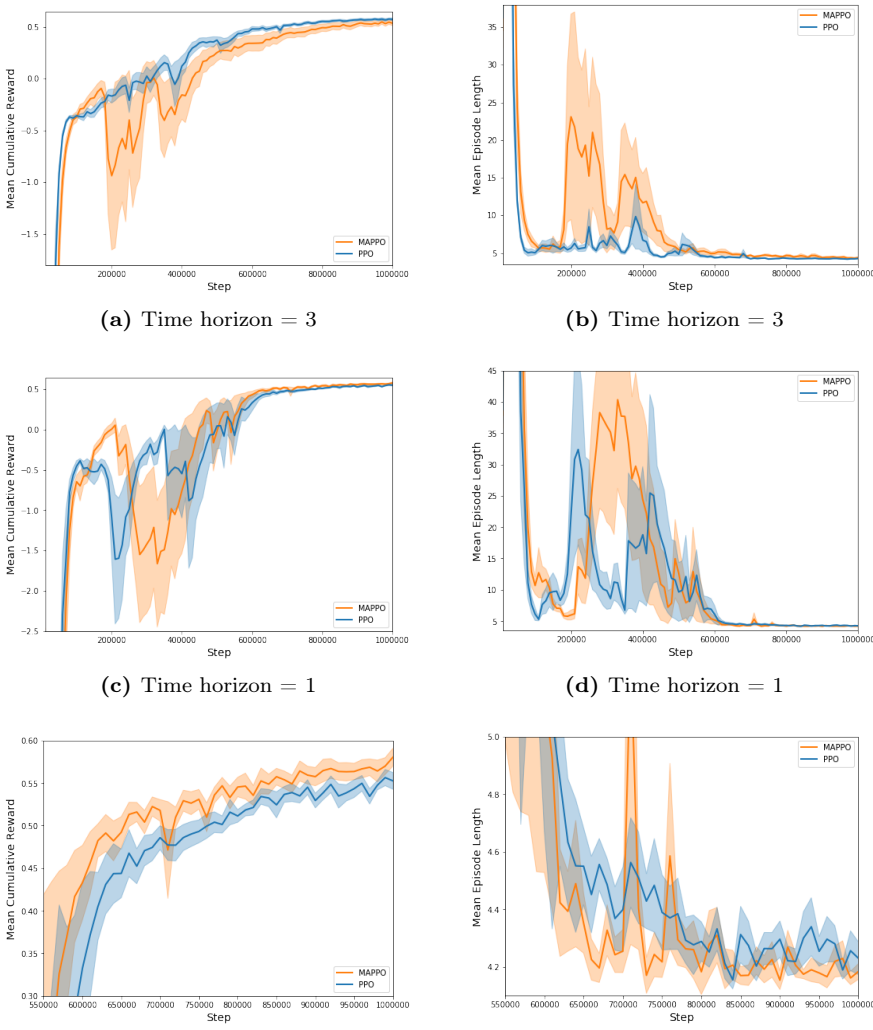
(a) The goal is the yellow pickup **(b)** The goal is the green pickup

Figure 4.6: Multi-Agent communication. Two agents, a speaker and a listener need to develop a language in order to be able to reach a goal that is different in every episode. The goal can either be collecting the yellow pickup, figure (a), collecting the green pickup, figure (b), or collecting the red pickup, figure(c)

plained earlier, when we pointed out the potential issues of applying traditional RL algorithms to multi-agent systems, PPO is able to converge and even reach a higher cumulative reward than MAPPO when the time horizon T is sufficiently large. One reason might be that, as mentioned in section 4.6, GAE [35] approximates the advantage function based on a combination of the MC sample returns and value function estimates. Although in PPO, the value function does take into account the influence of the other agents, the rewards given at every step are dependent on all the actions taken on the environment. Therefore, the more the PPO estimates rely on the MC sample returns the less biased the advantage function would be. This idea is supported by the figures 4.7c and 4.7d. When the time horizon $T = 1$, PPO has more difficulty to converge and is outperformed by MAPPO, whose critic can handle the non-stationarity introduced by having multiple agents, figures 4.7e and 4.7f

Tables with the hyperparameters used for training and other details about this experiment are provided in the appendix. Follow the link[5] to find a video showing the results of this experiment. As mentioned, because communication is not instantaneous, at the initial state of every episode, the listener receives a message that belongs to the previous one, which causes sometimes a change in its trajectory.

[5]Multi-Agent Communication: https://youtu.be/NWhmgi_liOc



(a) Time horizon = 3

(b) Time horizon = 3

(c) Time horizon = 1

(d) Time horizon = 1

(e) Time horizon = 1. Zoom

(f) Time horizon = 1. Zoom

Figure 4.7: Comparison between PPO and MAPPO algorithms on the multi-agent communication environment with 2 goals. The thick lines are the mean cumulative reward (a, c, e) and the mean episode length (b, d, f) per time step averaged over 10 runs. The light blue and orange shaded areas represent the one standard deviation confidence interval of the mean values. We used a random seed and the same hyperparameters for both algorithms. MAPPO outperforms PPO only for short time horizons.

4.6.3 Multi-Agent grid world: Learning to collaborate

The final example involves a predator-prey game. Three spaceships with limited vision, the predators, are chasing a fourth one, the prey, which moves randomly around the grid. The predators are penalized in every step while the prey remains alive. The predators receive a large reward when one of them captures the prey. Similar to [36], the reward is greater depending on whether only one, two or three of the spaceships are within the capture region. As in the previous example, the predators are also equipped with a communication channel so as to ease a collaborative strategy. They have therefore 6 actions available: stay still, move north, move south, move east, move west or send a message. The environment is depicted in figure 4.8 the prey is the orange spaceship facing upwards and the predators are the green, blue and brown spaceships facing downwards.

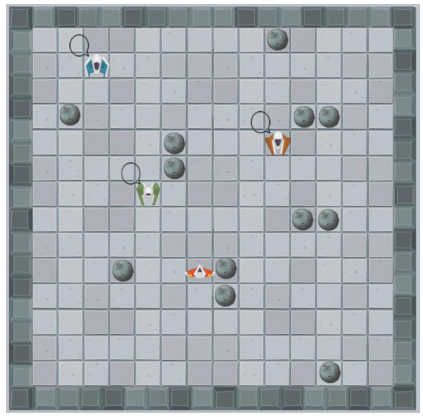
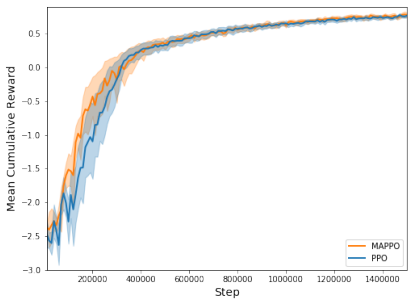


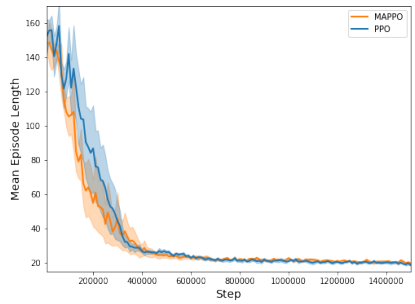
Figure 4.8: Multi-Agent cooperation. Three spaceships, the predators, with limited vision, are chasing a fourth one, the prey, which is trying to scape from them. The predators receive a greater reward if they capture the prey collectively.

The state observation of a predator consists of the coordinates of the other agents, their respective messages and the location of the prey if this is only 5 steps away from the predator. On the other hand, the capture region is limited to a radius of 3 cells from the prey.

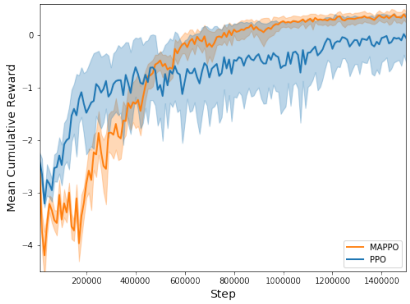
We again see in figures 4.9a and 4.9b that as long as the time horizon is kept large enough PPO and MAPPO are equally good at training teams of multiple agents. The problems arise when the sequence of partial rewards that is used in the advantage function is not long enough to cover for the biased value estimates. In that case and due to the intervention of the other agents, the non multi-agent



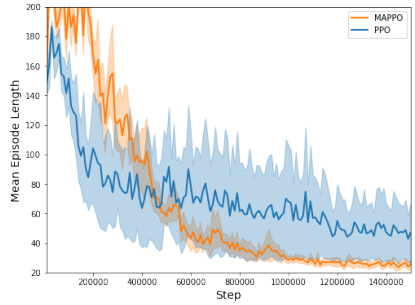
(a) Time horizon = 20



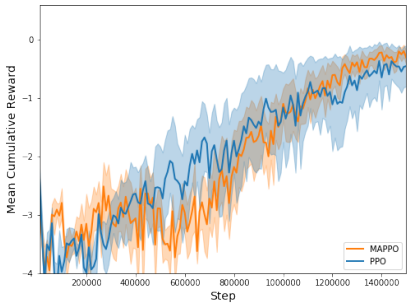
(b) Time horizon = 20



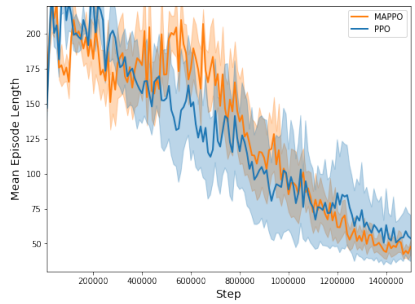
(c) Time horizon = 1



(d) Time horizon = 1



(e) Time horizon = 0

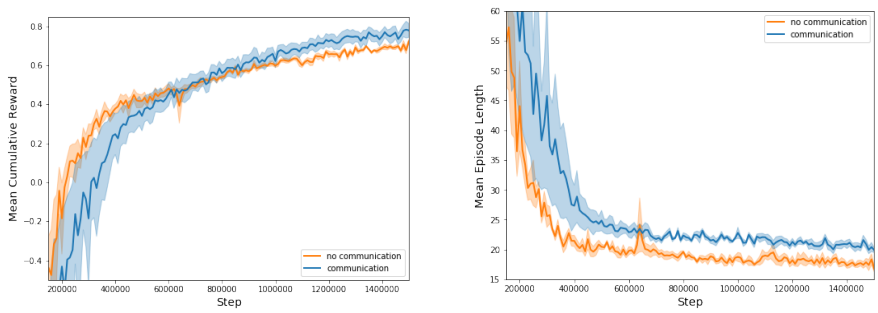


(f) Time horizon = 0

Figure 4.9: Multi-agent: predator-prey. The thick lines are the mean cumulative reward (a, c, e) and the mean episode length (b, d, f) per time step averaged over 5 runs. The light blue and orange shaded areas represent the one standard deviation confidence interval of the mean values. We used a random seed and the same hyperparameters for both algorithms. MAPPO outperforms PPO only for short time horizons where PPO is more instable and has more difficulties to converge.

methods are instable and seem to have more difficulties to converge. This is shown in the figures 4.9c, 4.9d, 4.9e and 4.9f where the one standard deviation confidence intervals of the blue curves are wider, the average returns are lower, and the episode lengths longer compared to the results obtained when training with the multi-agent method.

Contrarily to the previous example, where sending messages was essential for the agents to be able to solve the task, in this environment, there is no explicit need for communication. To evaluate whether or not it is actually helping the agents to coordinate and therefore find better policies, we blocked the communication channel and trained the agents again under the same conditions. The results are shown in figure 4.10. Despite having longer episode lengths, after 1.5M steps of training, the agents obtain higher returns on average. The reason being that when a predator finds the prey, if communication is allowed, it can call the others so that they can capture it together and receive a higher reward.



(a) **(b)**

Figure 4.10: Multi-agent predator-prey. The orange curves are the result of training agents that cant communicate using MAPPO. The blue curves are obtained training the agents, under the same conditions as in the previous case, but allowing them to send messages to each other. We see how communication helps coordination among the agents which are able to obtain higher returns on average.

One important limitation of RL is that the information is only available at the current time step. In this scenario, a message it is only seen at the state that comes immediately after the message is sent, meaning that the listeners quickly forget about what the speaker just said. One option to overcome this would be to have a memory slot and let the agents decide what information to store at each time step [37]. At expenses of having a larger state space and therefore requiring a bigger neural network the problem could be solved

by stacking various frames of observations. Finally, we could also make use of recurrent neural networks [38]. The agent generates a hidden state that is fed back into the system, therefore allowing it to keep track of what happened in the past.

Tables with the hyperparameters used for training and other details about this experiment are provided in the appendix. Follow the link[6] to find a video showing the results of this experiment. Note how when a predator sees the prey, it decides to alert the others so they can capture the prey together. The message is often sent multiple times due to the lack of memory.

[6]Multi-Agent Predator-Prey: https://youtu.be/cZXliRB2Gz8

Conclusion and Discussion

We started by introducing the concepts of Markov Decision Processes and Reinforcement Learning. We covered a variety of algorithms based on dynamic programming and policy iteration. We also showed that tabular methods work very well when having a reduced number of states and actions but they do not scale to large problems.

We demonstrated that tables can be substituted by parametric functions to obtain models that can better generalize over the state-action space. We presented the two main branches in Deep RL models, value based and policy based methods and compared the most successful algorithms in each family, DQN and PPO, using our own examples.

We explored how to avoid the non-stationary issues in transition and reward functions, when having more than one intelligent agent, by applying the Markov Games framework. We explained why selecting the correct partial rewards is crucial in multi-agent environments and how different reward functions might lead to situations of competitiveness or cooperation among agents.

We implemented two different decentralized methods for multi-agent reinforcement learning, MADQN and MAPPO, and compared them with the single agent algorithms presented in the third chapter. We showed that while the time horizon is long enough PPO can still learn a good policy but it starts to perform worse than MAPPO, the more the advantage estimates rely on the value function.

Finally we provided the agents with the ability to send and receive messages and showed in two different experiments how they can develop their own lan-

guage and use this messages to convey ideas, communicate and coordinate their actions.

The majority of the time and effort dedicated to this thesis has been directed towards implementing the RL algorithms, adapting Unity's code to accommodate our new functions and designing all the experiments and new environments in the editor.

In future work, alternatives to the multi-agent algorithms we implemented in this thesis could also be explored. Considering that, under certain conditions, agents might be allowed to share actions or observations during execution, we could study how much better centralized methods do, compared to independent policies.

On the other hand, a variant of the centralized critic method proposed in [39] could also be discussed. The algorithm is called Counterfactual Multi-Agent Policy Gradients (COMA) and it uses a baseline function that intends to measure how good the outcome of the chosen action is when compared to the others.

Because policies are stochastic, feeding one hot encoded vectors with the simulated actions of the other agents to the critic neural network can cause instabilities, especially at the beginning when all actions are equally likely to be chosen. We hypothesize that feeding the action probabilities instead might help reducing the initial oscillations and make the process more robust.

A common approach to speed up learning consists of planning ahead before taking an action using previous experiences. This is called model-based RL. The agent deploys its own model of the environment to simulate the process off-line and make more informed decisions on what actions to take [40]. One of the most successful techniques is Monte Carlo Tree Search [41], which assesses the value of unknown states in the tree by exploring random trajectories and back-propagating the sample returns.

Hierarchical learning [42] studies the possibility of grouping together states and actions with similar properties. These aggregations are called state and action abstractions and are very useful when combined with planning methods because they allow to reduce the tree size and thus ease simulations.

To substitute ϵ-greedy we could also incorporate other, more efficient, methods for exploration. In [43] they use Bayesian linear regression to distribute the probability of selecting any of the suboptimal actions according to the mean value estimate and the uncertainty.

Appendix

We tested the performance of our methods on a set of different environments. Because, the curves are normally very noisy, we ran each of the algorithms several times and took the mean value.

The hyperparameters were manually tuned in each example according to some general heuristic rules. We are aware that a proper parameter search is more appropriate when comparing different methods. Nonetheless, this normally requires a lot of time and computing power. Besides, the intention of this work was never to evaluate how good or bad one algorithm performed with respect to another but rather showing the advantages and limitations of each of them. For the same reason, we didn't test the performance of our methods on any of the popular benchmark examples. Instead, we preferred to design our own environments so as to be able to illustrate some of the theoretical concepts that are addressed in this thesis.

The tables below contain the hyperparameters that we used for training the agents and other details about our environments.

Static Grid World

Static Grid World	
Number of agents	1
Number of golden nuggets	1
Number of asteroids	1
Maximum episode length	50
Reward for collecting golden nugget	+1
Reward for colliding with asteroid	−1
Reward per step	−0.01

Static Grid World, tabular Q-learning		
Number of steps	5×10^3	Maximum number of training steps
gamma	0.99	Discount rate
ϵ_{start}	1	Initial probability of random action
ϵ_{end}	0	Final probability of random action
ϵ decay steps	4×10^3	Number of steps of linear decay for ϵ
α	0.8	Learning rate

Dynamic Grid World 1

Dynamic Grid World 1	
Number of agents	1
Number of golden nuggets	1
Number of asteroids	1
Maximum episode length	50
Reward for collecting golden nugget	+1
Reward for colliding with asteroid	−1
Reward per step	−0.05

Dynamic Grid World 1, DQN		
Number of steps	5×10^5	Maximum number of training steps
γ	0.99	Discount rate
ϵ start	1	Initial probability of random action
ϵ end	0	Final probability of random action
ϵ decay steps	3×10^5	Number of steps of linear decay for ϵ
α	3×10^{-3}	Learning rate
Memory size	5×10^4	Replay memory maximum size
Pre-train steps	1×10^6	Steps random action before first update
Batch size	32	Number of experiences in every update
Number of layers	2	Number of hidden layers in the network
Number of units	32	Number of units in each hidden layer
Update frequency	4	Number of steps between model updates
τ	1×10^{-3}	Update rate of target model

Dynamic Grid World 1, PPO		
Number of steps	5×10^5	Maximum number of training steps
γ	0.99	Reward discount rate
λ	0.95	GAE discount rate
α	3×10^{-3}	Learning rate
ϵ	0.2	Divergence between old and new policies
β	1×10^{-3}	Strength of entropy regularization
Number of epoch	5	Passes through buffer per update
Batch size	32	Number of experiences in every update
Buffer size	2048	Size of the memory buffer before update
Number of layers	2	Number of hidden layers in the network
Number of units	32	Number of units in each hidden layer
Time horizon	1	Number of steps GAE calculation

Dynamic Grid World 2

Dynamic Grid World 2	
Number of agents	1
Number of golden nuggets	2
Number of asteroids	2
Maximum episode length	100
Reward for collecting golden nugget	+1
Reward for colliding with asteroid	−1
Reward per step	−0.05

Dynamic Grid World 2, DQN		
Number of steps	5×10^5	Maximum number of training steps
γ	0.99	Discount rate
ϵ start	1	Initial probability of random action
ϵ end	0	Final probability of random action
ϵ decay steps	3×10^5	Number of steps of linear decay for ϵ
α	3×10^{-3}	Learning rate
Memory size	5×10^4	Replay memory maximum size
Pre-train steps	1×10^4	Steps random action before first update
Batch size	32	Number of experiences in every update
Number of layers	2	Number of hidden layers in the network
Number of units	32	Number of units in each hidden layer
Update frequency	4	Number of steps between model updates
τ	1×10^{-3}	Update rate of target model

Dynamic Grid World 2, PPO		
Number of steps	5×10^5	Maximum number of training steps
γ	0.99	Reward discount rate
λ	0.95	GAE discount rate
α	3×10^{-3}	Learning rate
ϵ	0.2	Divergence between old and new policies
β	1×10^{-3}	Strength of entropy regularization
Number of epoch	5	Passes through buffer per update
Batch size	32	Number of experiences in every update
Buffer size	2048	Size of the memory buffer before update
Number of layers	2	Number of hidden layers in the network
Number of units	32	Number of units in each hidden layer
Time horizon	1	Number of steps GAE calculation

Multi-Agent Competition 1

Multi-Agent Competition 1	
Number of agents	2
Number of golden nuggets	1
Number of asteroids	2
Maximum episode length	100
Individual reward for collecting golden nugget	+1
Individual reward for colliding with asteroid	−1
Individual reward per step	−0.05

Multi-Agent Competition 1, DQN		
Number of steps	1×10^6	Maximum number of training steps
γ	0.99	Discount rate
ϵ start	1	Initial probability of random action
ϵ end	0	Final probability of random action
ϵ decay steps	3×10^5	Number of steps of linear decay for ϵ
α	2×10^{-3}	Learning rate
Memory size	5×10^4	Replay memory maximum size
Pre-train steps	1×10^4	Steps random action before first update
Batch size	128	Number of experiences in every update
Number of layers	3	Number of hidden layers in the network
Number of units	64	Number of units in each hidden layer
Update frequency	4	Number of steps between model updates
τ	1×10^{-3}	Update rate of target model

Multi-Agent Competition 1, MADQN		
Number of steps	1×10^6	Maximum number of training steps
γ	0.99	Discount rate
ϵ start	1	Initial probability of random action
ϵ end	0	Final probability of random action
ϵ decay steps	3×10^5	Number of steps of linear decay for ϵ
α	2×10^{-3}	Learning rate
Memory size	5×10^4	Replay memory maximum size
Pre-train steps	1×10^4	Steps random action before first update
Batch size	128	Number of experiences in every update
Number of layers	3	Number of hidden layers in the network
Number of units	64	Number of units in each hidden layer
Update frequency	4	Number of steps between model updates
τ	1×10^{-3}	Update rate of target model
Update frozen frequency	2×10^5	Number of steps frozen agent updates

Multi-Agent Competition 2

Multi-Agent Competition 2	
Number of agents	2
Number of golden nuggets	2
Number of asteroids	2
Maximum episode length	100
Individual reward for collecting golden nugget	+1
Individual reward for colliding with asteroid	−1
Individual reward per step	−0.05

Multi-Agent Competition 2, DQN		
Number of steps	1×10^6	Maximum number of training steps
γ	0.99	Discount rate
ϵ start	1	Initial probability of random action
ϵ end	0	Final probability of random action
ϵ decay steps	3×10^5	Number of steps of linear decay for ϵ
α	2×10^{-3}	Learning rate
Memory size	5×10^4	Replay memory maximum size
Pre-train steps	1×10^4	Steps of random action before first update
Batch size	128	Number of experiences in every update
Number of layers	3	Number of hidden layers in the network
Number of units	64	Number of units in each hidden layer
Update frequency	4	Number of steps between model updates
τ	1×10^{-3}	Update rate of target model

Multi-Agent Competition 2, MADQN		
Number of steps	1×10^6	Maximum number of training steps
γ	0.99	Discount rate
ϵ start	1	Initial probability of random action
ϵ end	0	Final probability of random action
ϵ decay steps	3×10^5	Number of steps of linear decay for ϵ
α	2×10^{-3}	Learning rate
Memory size	5×10^4	Replay memory maximum size
Pre-train steps	1×10^4	Steps of random action before first update
Batch size	128	Number of experiences in every update
Number of layers	3	Number of hidden layers in the network
Number of units	64	Number of units in each hidden layer
Update frequency	4	Number of steps between model updates
τ	1×10^{-3}	Update rate of target model
Update frozen frequency	2×10^5	Number of steps frozen agent updates

Multi-Agent Cooperation 1

Multi-Agent Cooperation 1	
Number of agents	2
Number of golden nuggets	1
Number of asteroids	2
Maximum episode length	100
Collective reward for collecting golden nugget	+1
Collective reward for colliding with asteroid	−1
Individual reward per step	−0.05

Multi-Agent Cooperation 1, MADQN		
Number of steps	1×10^6	Maximum number of training steps
γ	0.99	Discount rate
ϵ start	1	Initial probability of random action
ϵ end	0	Final probability of random action
ϵ decay steps	3×10^5	Number of steps of linear decay for ϵ
α	2×10^{-3}	Learning rate
Memory size	5×10^4	Replay memory maximum size
Pre-train steps	1×10^4	Steps random action before first update
Batch size	128	Number of experiences in every update
Number of layers	3	Number of hidden layers in the network
Number of units	64	Number of units in each hidden layer
Update frequency	4	Number of steps between model updates
τ	1×10^{-3}	Update rate of target model
Update frozen frequency	2×10^5	Number of steps frozen agent updates

Multi-Agent Cooperation 1, PPO		
Number of steps	1×10^6	Maximum number of training steps
γ	0.99	Reward discount rate
λ	0.95	GAE discount rate
α	2×10^{-3}	Learning rate
ϵ	0.2	Divergence between old and new policies
β	1×10^{-3}	Strength of entropy regularization
Number of epoch	5	Passes through buffer per update
Batch size	128	Number of experiences in every update
Buffer size	2048	Size of the memory buffer before update
Number of layers	3	Number of hidden layers in the network
Number of units	64	Number of units in each hidden layer
Time horizon	$1,0$	Number of steps GAE calculation

Multi-Agent Cooperation 1, MAPPO		
Number of steps	1×10^6	Maximum number of training steps
γ	0.99	Reward discount rate
λ	0.95	GAE discount rate
α	2×10^{-3}	Learning rate
ϵ	0.2	Divergence between old and new policies
β	1×10^{-3}	Strength of entropy regularization
Number of epoch	5	Passes through buffer per update
Batch size	128	Number of experiences in every update
Buffer size	2048	Size of the memory buffer before update
Number of layers	3	Number of hidden layers in the network
Number of units	64	Number of units in each hidden layer
Time horizon	$1,0$	Number of steps GAE calculation

Multi-Agent Cooperation 2

Multi-Agent Cooperation 2	
Number of agents	2
Number of golden nuggets	2
Number of asteroids	2
Maximum episode length	100
Collective reward for collecting golden nugget	$+1$
Collective reward for colliding with asteroid	-1
Individual reward per step	-0.05

Multi-Agent Cooperation 2, MADQN		
Number of steps	1×10^6	Maximum number of training steps
γ	0.99	Discount rate
ϵ start	1	Initial probability of random action
ϵ end	0	Final probability of random action
ϵ decay steps	3×10^5	Number of steps of linear decay for ϵ
α	2×10^{-3}	Learning rate
Memory size	5×10^4	Replay memory maximum size
Pre-train steps	1×10^4	Steps random action before first update
Batch size	128	Number of experiences in every update
Number of layers	3	Number of hidden layers in the network
Number of units	64	Number of units in each hidden layer
Update frequency	4	Number of steps between model updates
τ	1×10^{-3}	Update rate of target model
Update frozen frequency	2×10^5	Number of steps frozen agent updates

Multi-Agent Cooperation 2, PPO		
Number of steps	1×10^6	Maximum number of training steps
γ	0.99	Reward discount rate
λ	0.95	GAE discount rate
α	2×10^{-3}	Learning rate
ϵ	0.2	Divergence between old and new policies
β	1×10^{-3}	Strength of entropy regularization
Number of epoch	5	Passes through buffer per update
Batch size	128	Number of experiences in every update
Buffer size	2048	Size of the memory buffer before update
Number of layers	3	Number of hidden layers in the network
Number of units	64	Number of units in each hidden layer
Time horizon	$2, 1$	Number of steps GAE calculation

Multi-Agent Cooperation 2, MAPPO		
Number of steps	1×10^6	Maximum number of training steps
γ	0.99	Reward discount rate
λ	0.95	GAE discount rate
α	2×10^{-3}	Learning rate
ϵ	0.2	Divergence between old and new policies
β	1×10^{-3}	Strength of entropy regularization
Number of epoch	5	Passes through buffer per update
Batch size	128	Number of experiences in every update
Buffer size	2048	Size of the memory buffer before update
Number of layers	3	Number of hidden layers in the network
Number of units	64	Number of units in each hidden layer
Time horizon	$2, 1$	Number of steps GAE calculation

Multi-Agent Communication

Multi-Agent Communication	
Number of agents	2
Number of golden nuggets	2
Number of asteroids	2
Maximum episode length	100
Collective reward for collecting golden nugget	$+1$
Individual reward for colliding with asteroid	-1
Individual reward per step	-0.05

Multi-Agent Communication, PPO		
Number of steps	1×10^6	Maximum number of training steps
γ	0.99	Reward discount rate
λ	0.95	GAE discount rate
α	2×10^{-3}	Learning rate
ϵ	0.2	Divergence between old and new policies
β	1×10^{-3}	Strength of entropy regularization
Number of epoch	5	Passes through buffer per update
Batch size	128	Number of experiences in every update
Buffer size	2048	Size of the memory buffer before update
Number of layers	3	Number of hidden layers in the network
Number of units	64	Number of units in each hidden layer
Time horizon	$3, 1$	Number of steps GAE calculation

Multi-Agent Communication, MAPPO		
Number of steps	1×10^6	Maximum number of training steps
γ	0.99	Reward discount rate
λ	0.95	GAE discount rate
α	2×10^{-3}	Learning rate
ϵ	0.2	Divergence between old and new policies
β	1×10^{-3}	Strength of entropy regularization
Number of epoch	5	Passes through buffer per update
Batch size	128	Number of experiences in every update
Buffer size	2048	Size of the memory buffer before update
Number of layers	3	Number of hidden layers in the network
Number of units	64	Number of units in each hidden layer
Time horizon	$3, 1$	Number of steps GAE calculation

Multi-Agent Predator-Prey

Multi-Agent Predator-Prey	
Number of predators	3
Number of preys	1
Maximum episode length	200
Individual reward per step	-0.02
Collective reward for capturing the prey	
1 agent within capture region	$+0.5$
2 agents within capture region	$+1$
3 agents within capture region	$+1.5$

Multi-Agent Predator-Prey, PPO		
Number of steps	1.5×10^6	Maximum number of training steps
γ	0.99	Reward discount rate
λ	0.95	GAE discount rate
α	3×10^{-3}	Learning rate
ϵ	0.2	Divergence between old and new policies
β	2×10^{-3}	Strength of entropy regularization
Number of epoch	5	Passes through buffer per update
Batch size	128	Number of experiences in every update
Buffer size	2048	Size of the memory buffer before update
Number of layers	3	Number of hidden layers in the network
Number of units	64	Number of units in each hidden layer
Time horizon	$20, 1, 0$	Number of steps GAE calculation

Multi-Agent Predator-Prey, MAPPO		
Number of steps	1.5×10^6	Maximum number of training steps
γ	0.99	Reward discount rate
λ	0.95	GAE discount rate
α	3×10^{-3}	Learning rate
ϵ	0.2	Divergence between old and new policies
β	2×10^{-3}	Strength of entropy regularization
Number of epoch	5	Passes through buffer per update
Batch size	128	Number of experiences in every update
Buffer size	2048	Size of the memory buffer before update
Number of layers	3	Number of hidden layers in the network
Number of units	64	Number of units in each hidden layer
Time horizon	$20, 1, 0$	Number of steps GAE calculation

Bibliography

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[3] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge, 1998.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[5] H. Gintis, *Moral sentiments and material interests: The foundations of cooperation in economic life*, vol. 6. MIT press, 2005.

[6] S. J. Brams, *Game theory and politics*. Courier Corporation, 2011.

[7] T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch, "Emergent complexity via multi-agent competition," *arXiv preprint arXiv:1710.03748*, 2017.

[8] A. Tampuu, T. Matiisen, D. Kodelja, I. Kuzovkin, K. Korjus, J. Aru, J. Aru, and R. Vicente, "Multiagent cooperation and competition with deep reinforcement learning," *PloS one*, vol. 12, no. 4, p. e0172395, 2017.

[9] M. Tan, "Multi-agent reinforcement learning: Independent vs. cooperative agents," in *Proceedings of the tenth international conference on machine learning*, pp. 330–337, 1993.

[10] R. Lowe, Y. Wu, A. Tamar, J. Harb, O. P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," in *Advances in Neural Information Processing Systems*, pp. 6382–6393, 2017.

[11] J. Foerster, I. A. Assael, N. de Freitas, and S. Whiteson, "Learning to communicate with deep multi-agent reinforcement learning," in *Advances in Neural Information Processing Systems*, pp. 2137–2145, 2016.

[12] R. E. Bellman and S. E. Dreyfus, *Applied dynamic programming*, vol. 2050. Princeton university press, 2015.

[13] S. P. Singh and R. S. Sutton, "Reinforcement learning with replacing eligibility traces," *Machine learning*, vol. 22, no. 1-3, pp. 123–158, 1996.

[14] C. Bishop, C. M. Bishop, *et al.*, *Neural networks for pattern recognition*. Oxford university press, 1995.

[15] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[16] A. Juliani, *Simple Reinforcement Learning with Tensorflow*, 2016 (accessed July 22, 2018). https://medium.com/@awjuliani.

[17] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, vol. 8, no. 3-4, pp. 293–321, 1992.

[18] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning.," in *AAAI*, vol. 16, pp. 2094–2100, 2016.

[19] H. V. Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems*, pp. 2613–2621, 2010.

[20] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, pp. 1057–1063, 2000.

[21] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE transactions on systems, man, and cybernetics*, no. 5, pp. 834–846, 1983.

[22] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," in *Reinforcement Learning*, pp. 5–32, Springer, 1992.

[23] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[24] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International Conference on Machine Learning*, pp. 1889–1897, 2015.

[25] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[26] L. Busoniu, R. Babuska, and B. De Schutter, "A comprehensive survey of multiagent reinforcement learning," *IEEE Trans. Systems, Man, and Cybernetics, Part C*, vol. 38, no. 2, pp. 156–172, 2008.

[27] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Machine Learning Proceedings 1994*, pp. 157–163, Elsevier, 1994.

[28] M. J. Osborne *et al.*, *An introduction to game theory*, vol. 3. Oxford university press New York, 2004.

[29] J. Hu and M. P. Wellman, "Nash q-learning for general-sum stochastic games," *Journal of machine learning research*, vol. 4, no. Nov, pp. 1039–1069, 2003.

[30] M. Bowling, "Convergence problems of general-sum multiagent reinforcement learning," in *ICML*, pp. 89–94, 2000.

[31] M. L. Littman, "Friend-or-foe q-learning in general-sum games," in *ICML*, vol. 1, pp. 322–328, 2001.

[32] F. A. Oliehoek, "Decentralized pomdps," in *Reinforcement Learning*, pp. 471–503, Springer, 2012.

[33] R. Nair, M. Tambe, M. Yokoo, D. Pynadath, and S. Marsella, "Taming decentralized pomdps: Towards efficient policy computation for multiagent settings," in *IJCAI*, vol. 3, pp. 705–711, 2003.

[34] M. Egorov, "Multi-agent deep reinforcement learning," 2016.

[35] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

[36] J. Z. Leibo, V. Zambaldi, M. Lanctot, J. Marecki, and T. Graepel, "Multi-agent reinforcement learning in sequential social dilemmas," in *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pp. 464–473, International Foundation for Autonomous Agents and Multiagent Systems, 2017.

[37] I. Mordatch and P. Abbeel, "Emergence of grounded compositional language in multi-agent populations," *arXiv preprint arXiv:1703.04908*, 2017.

[38] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," *CoRR, abs/1507.06527*, vol. 7, no. 1, 2015.

[39] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson, "Counterfactual multi-agent policy gradients," *arXiv preprint arXiv:1705.08926*, 2017.

[40] R. S. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *Machine Learning Proceedings 1990*, pp. 216–224, Elsevier, 1990.

[41] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *International conference on computers and games*, pp. 72–83, Springer, 2006.

[42] L. P. Kaelbling, "Hierarchical learning in stochastic domains: Preliminary results," in *Proceedings of the tenth international conference on machine learning*, vol. 951, pp. 167–173, 1993.

[43] K. Azizzadenesheli, E. Brunskill, and A. Anandkumar, "Efficient exploration through bayesian deep q-networks," *arXiv preprint arXiv:1802.04412*, 2018.