Alma Mater Studiorum · University of Bologna

Faculty of Science
Laurea Magistrale in Computer Science

FLATLAND:
A study of Deep Reinforcement Learning methods applied to the vehicle rescheduling problem in a railway environment

Supervisor:
Prof. Andrea Asperti

Author:
Giulia Cantini

Co-supervisor:
Dr. Andrew Melnik

Co-supervisor:
Prof. Helge Ritter

Session III
Academic Year 2018/2019

To my grandparents

Abstract

In Reinforcement Learning l'obiettivo è apprendere come degli agenti debbano prendere sequenze di decisioni o azioni all'interno di un ambiente al fine di massimizzare un valore numerico chiamato rinforzo (*reward*). Questo processo di apprendimento impiegato in combinazione con reti neurali artificiali ha dato origine al Deep Reinforcement Learning (DRL), applicato ad oggi a molti domini, a partire dai videogiochi alla robotica e ai veicoli autonomi.

Questo elaborato investiga dei possibili approcci che utilizzano DRL applicandoli a *Flatland*, una simulazione ferroviaria multi-agente in cui l'obiettivo principale è pianificare i percorsi dei treni in modo da ottimizzare il flusso di traffico all'interno della rete e riprogrammarli in caso di eventi che ne ostacolino il cammino. Il problema all'origine dei task proposti in Flatland è il *Vehicle Rescheduling Problem*, un problema NP-completo di ottimizzazione combinatoria, per il quale l'impiego di euristiche e metodi deterministici per identificare soluzioni subottimali, non è efficace in sistemi ferroviari realistici.

In particolare, si sono analizzati in questo ambiente il task della *navigazione* di un singolo agente all'interno della mappa, che a partire da una posizione iniziale deve raggiungere una stazione target nel minor tempo possibile; e la generalizzazione di questo task ad un sistema multi-agente, in cui al problema della navigazione si aggiunge quello della risoluzione dei conflitti tra agenti, i cui percorsi dalla sorgente alla destinazione possono potenzialmente incrociarsi.

Per risolvere il problema si sono sviluppate delle *osservazioni* specifiche dell'ambiente, in modo da catturare le informazioni necessarie per la rete,

addestrata con Deep Q-Learning e varianti, ad apprendere come decidere per ogni agente l'azione che porti alla soluzione migliore, quella che massimizza il reward totale.

I risultati positivi ottenuti su environment semplificati danno spazio a numerose interpretazioni e possibili sviluppi futuri e mostrano come il Reinforcement Learning abbia le potenzialità per risolvere il problema in una nuova prospettiva.

Acknowledgements

I would like to thank my supervisors, Dr. Andrew Melnik, for his guidance during the development of this work and his exceptional availability in brainstorming with me any possible ideas, and Prof. Andrea Asperti for the experience and insight shared and the time dedicated to the progress of this research. I'm also grateful to my colleague Devid Farinelli for his precious collaboration in improving and extending this project.
Lastly, I would like to thank my family that has been supporting me throughout my studies and my dear friends, invaluable source of joy.

Introduction

Reinforcement learning (RL) is a subfield in machine learning, where the task is learning how agents should take sequences of actions in an environment in order to maximize a numerical reward signal [1] [5]. Over the past few years, RL has become increasingly popular, mainly thanks to the achievements obtained in combination of deep learning techniques, that gave birth to the new field of Deep Reinforcement Learning (DRL).
Among the most notable works that contributed to donate to DRL significant recognition worldwide there are DeepMind papers on Atari games [7] [8], Go [9] [10] and other board games [11] [12]. Apart from games, Deep Reinforcement Learning is applied on a variety of disciplines such as robotics and autonomous vehicles [13] [14] [15], natural language processing [16] [17], computer vision [18] and many others [6], where it has shown potential and has been successful in solving many complex decision-making tasks that were previously out of reach for a machine. In this work, we investigate the application of Deep Reinforcement Learning techniques in the real-world scenario of transportation, with the aim of solving the combinatorial optimization problem of vehicle re-scheduling (VRSP) [19] in a multi-agent railway environment called *Flatland*. The goal in this simulation is to optimize traffic flow inside a network: a research focus particularly interesting, since possible advancements could lead to improvements in the way modern traffic management systems deal with vehicles, in many areas of logistics and transportation.

Classical solutions to the VRSP, a NP-complete problem in combinatorial optimization, are based on the use of heuristics and other deterministic approaches to find suboptimal but acceptable solutions. However, these solutions are only applicable to a simplified railway simulation such as Flatland, where many of the real-world constraints were removed, but in a real-world scenario with realistic maps and a fast-changing environment, heuristics have been proved to be insufficient to obtain feasible solutions. To be applicable to real railway systems, an offloading of the computational effort of these approaches to a *learning component* could be potentially beneficial to solve the problem efficiently.
It is in this perspective, that we propose the methods in this work, to overcome the limits of the classical approaches and identify new alternatives in the field of Reinforcement Learning.

Chapter 1 contains a theoretical introduction on RL in its definition in relationship to Markov Decision Processes (MDPs), it describes the main ideas and elements, such as the concepts of *reward*, *policy* and *value function*, the foundational algorithmic techniques to solve it, namely Monte Carlo methods, Dynamic Programming and Temporal-Difference Learning and explores in the end the differences with the variant of Multi-Agent Reinforcement Learning (MARL).

Chapter 2 focuses on the results obtained combining one of the algorithm presented in Chapter 1 (Q-Learning), with artificial neural networks, by giving a detailed overview of the improvements and the achievements over time, starting from the vanilla DQN to Rainbow, detailing in sections the single relevant modifications.

Chapter 3 gives a thorough explanation of the problem we are trying to solve, describing the Flatland environment, its core parts, with a perspective on the possible tasks and the potential of this simulated system.

Chapter 4 describes the research work proposed to solve the tasks in the Flatland environment, from finding a suitable policy in the navigation task to the development of new observation classes to tackle the problem of con-

flicts avoidance in a multi-agent setting.

Chapter 5 briefly summarizes the results of the approaches presented, highlighting the major difficulties posed by the challenge, and further suggests possible improvements as a future work.

Contents

List of Figures

List of Tables

Chapter 1

Background

As human beings, the principle of learning by interacting with the surrounding environment accompanies us in our lives since our first experiences as infants.
When we walk, look and talk, we are supported by a complex sensorimotor system that allows us to perceive the world and change it with our actions and behavior.
Throughout our existences, such interactions are of undoubted importance since they concur in building a significant amount of experiences about ourselves and the world, that will translate into knowledge.
This idea of learning by interaction is indeed foundational in nearly all theories of learning and intelligence.

In this chapter we explore *reinforcement learning*, a computational approach to goal-directed learning by interaction, that focuses on providing solutions to teach machines and artificial intelligence to act and learn in an environment, exactly as a human being.
First we define it mathematically in terms of a Markov Decision Process, then we provide an overview of all its meaningful elements, finally explaining some of the first approaches studied in literature to solve the RL problem, such as Dynamic Programming, Monte Carlo and Temporal Difference learning.

1.1 Reinforcement learning

Reinforcement learning is learning what to do - how to map situations to actions - so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them [1]. In most situations, actions may influence not only immediate rewards, but also the consequent situations and through those, also subsequent rewards. These two characteristics of *trial-and-error* search and *delayed reward* define reinforcement learning and distinguish it in the machine learning field.
Reinforcement learning is usually formalized using ideas from dynamical systems theory, in particular Markov decision processes. These processes are well-suited to define the agent-environment interaction as intended to include the three aspects of *sensation*, *action*, and *goal*.
RL is different from *supervised learning*, since it is not based on a training set of labeled data provided by some external supervisor.
It is also different from *unsupervised learning*, since its goal is to maximize a reward signal, not to find hidden structure inside collections of unlabeled data.
Another aspect that characterizes reinforcement learning is the trade-off between exploration and exploitation, that is the choice between trying actions with unknown outcomes to see how they affect the environment or perform actions with expected outcome according to the knowledge already available. The dilemma lies in the fact that, to obtain the best performance, the agent has to *exploit* what it has already experienced in order to obtain reward, but it also has to *explore* in order to make better action selections in the future. Furthermore, one of the most interesting features of reinforcement learning, is its interactions with many other engineering and scientific disciplines, that are, as many subfields in artificial intelligence, statistics, optimization, op-

erations research, control theory and other mathematical subjects, but also psychology, biology and more precisely neuroscience. Reinforcement learning has received from these fields the concept of how animals learn to fulfill a goal and many ideas behind some of the most important algorithms, that were inspired by biological learning systems.

1.2 Finite Markov Decision Processes (MDP)

In this section we introduce finite Markov decision processes as a suitable formalism to define sequential decision making, as in the reinforcement learning framework.

1.2.1 Agent-Environment Interaction

In order to exploit the MDP formalism, we first frame the problem in an interaction scheme between a learner, called *agent* and the outside on which the agent can act, the *environment*. In particular, this interaction occurs in a sequence of discrete time steps $t = 0, 1, 2, 3, ...$ where at each time step t, the agent *observes* the environment, receiving some representation of its *state* $S_t \in \mathcal{S}$, and upon that, selects an *action*, $A_t \in \mathcal{A}(s)$, from the set of all possible actions. At the following time step, the agent receives a numerical reward $R_{t+1} \in \mathcal{R}$ and enters a new state, S_{t+1}.
The mechanism is detailed in Figure 1.1.

In a finite MDP, the sets of states, actions, and rewards ($\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$) have a finite number of elements. In addition, the random variables R_t and S_t have discrete probability distributions that depend only on the previous state and action, following the Markov property.

Definition 1.2.1. *A discrete time stochastic control process is Markovian*

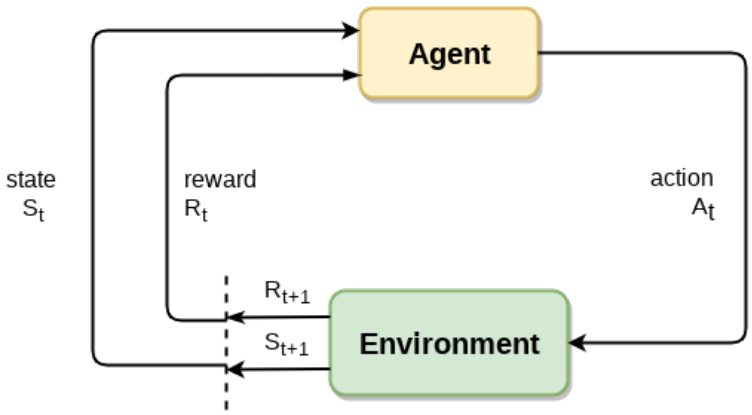


Figure 1.1: Agent interacting in an environment in the RL framework.

(i.e. it has the Markov property) if

$$P(S_{t+1} \mid S_t) = P(S_{t+1} \mid S_1, ..., S_t). \tag{1.1}$$

A Markov Decision Process [20] is a discrete time stochastic control process defined as follows:

Definition 1.2.2. *A (finite) MDP is a 5-tuple* $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ *where:*

- $\mathcal{S}$ *is a (finite) set of states,*
- $\mathcal{A}$ *is a (finite) set of actions,*
- $\mathcal{P}$ *is a state transition probability function* $P : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ $p(s' \mid s, a) = P(S_t = s' \mid S_{t-1} = s, A_{t-1} = a)$ *(set of conditional probabilities between states).*
- $\mathcal{R}$ *is the reward function* $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ $R(s, a) = \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \mid s, a).$
- γ *is a discount factor* $\gamma \in [0, 1]$.

1.2.2 Reward

In reinforcement learning the goal is formalized through the idea of *reward signal*; at each time step, the agent receives a numeric value, $R_t \in \mathbb{R}$ and its goal is to maximize the total amount of reward received, intended not as *immediate reward* but as sum of rewards in the long run.
This idea is known as *reward hypothesis*:

That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward) [1].

The way we shape reward directly influences the behaviour of our agent, that is why reward definition is a key point when defining a reinforcement learning model.

1.2.3 Returns and Episodes

To express more formally this idea of cumulative reward it is necessary to introduce the concept of *expected return*, denoted as G_t, as a function of the sequence of rewards, that we seek to maximize.
In the simplest case

$$G_t = R_{t+1} + R_{t+2} + ... + R_T, \tag{1.2}$$

where T is the final step.
However, this definition applies well only in presence of *episodic tasks*, namely when the agent-environment interaction can be naturally divided into subsequences, called *episodes*, where each of them ends in a special *terminal state*. On the other hand, in presence of *continuing tasks* where the interaction goes on without limits and the sequence of states is impossible to divide, the return formulation given is problematic, since for $T = \infty$ would lead to infinite reward.

For this reason, the commonly accepted formulation of return makes use of the definition of *discount*:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\inf} \gamma^k R_{t+k+1}, \tag{1.3}$$

where γ is a parameter $\in [0, 1]$ called *discount rate*.

The discount rate estimates the present value of future rewards, in other words, it works as a "weight" on the importance of future rewards, where a reward received k steps in the future is worth only γ^{k+1} times what it would be worth if received at the current time step.
A value of $\gamma < 1$ grants convergence of the sequence to a finite value, given that R_k is bounded.
If $\gamma = 0$ the agent "sees" only one step ahead, trying to maximize only immediate rewards R_{t+1} . If $\gamma = 1$ the agent considers all future rewards as worthy as the immediate ones.
In general, for a value of γ approaching 1, the agent takes into account future rewards more strongly, in other words it becomes more farsighted.

1.2.4 Policies and Value Functions

A *value function* is a function of states or state-action pairs, that estimates how good is for an agent to be in a given state, defined in terms of future expected return. This value function is evaluated against a policy, that is informally, the agent's behaviour.
Formally, a *policy* π is a probability distribution over actions $a \in \mathcal{A}(S)$ for each $s \in \mathcal{S}$ where

$$\pi(a \mid s) = p(A_t = a \mid S_t = s) \tag{1.4}$$

in other words, the probability of selecting a possible action given a state. The *value* of a state s under a policy π, denoted v_s, is the expected return when starting in s and following π thereafter. Formally:

$$v_\pi(s) = \mathbb{E}[G_t \mid S_t = s] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right], \text{ for all } s \in S, \quad (1.5)$$

where $\mathbb{E}_\pi[.]$ denotes the expected value of a random variable given that the agent follows policy π, and t is a time step.
This v_π is called the *state-value function for policy* π.
In a similar manner we can define the value of selecting an action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a and then following policy π:

$$q_\pi(s, a) = \mathbb{E}[G_t \mid S_t = s, A_t = a] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right]. \quad (1.6)$$

Here q_π is called the *action value function for policy* π.
Lastly, it is possible to define a third value function, called *advantage function*:

$$a_\pi(s, a) = q_\pi(s, a) - v_\pi(s) \quad (1.7)$$

that uses both the *action value function* q and the value function v to describe how good an action a is compared to the expected return when following a policy π.

1.2.5 Bellman Equation

A fundamental feature of value functions is that they satisfy recursive relationships known as *Bellman equations*.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] && (1.8)\\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] && (1.9)\\ &= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a)\Big[r + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']\Big] && (1.10)\\ &= \sum_a \pi_(a \mid s) \sum_{s', r} p(s', r \mid s, a)\Big[r + \gamma v_\pi(s')\Big], \text{ for all } s \in \mathcal{S}, && (1.11) \end{aligned}$$

where $a \in \mathcal{A}$, $s' \in \mathcal{S}$ and $r \in \mathcal{R}$.

The Bellman equation expresses a relationship between the value of a state and the value of its successor states.

This idea can be exemplified through the backup diagram in Figure 1.2.

An open circle represents a state and a solid one represents a state-action pair. Starting from state s, the root node of the tree, the agent can select an action following policy π. From each of these action the environment can respond with one of the several next states s' along with the correspondent reward, according to the distribution p.

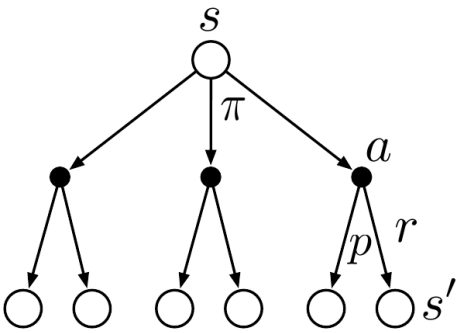


Figure 1.2: Backup diagram for v_π

The Bellman equation averages over all the possibilities, weighting each by its probability of occurring. In other words, it says that the value of the start state equals the value of the expected next state, plus the reward expected along the way.

This equation has a unique solution, the value function v_π. The Bellman equation is very interesting since it provides a basis to compute, approximate and learn v_π.

1.2.6 Optimality

The goal in a reinforcement learning task is to identify the policy that achieves the most reward in the long run.

To define an *optimal policy* however, it is first fundamental to define a partial ordering over policies. A policy π is better than or equal to a policy π' if its expected return is greater than or equal to that of π' in all states. Formally:

$$\pi \geq \pi' \text{ if and only if } v_\pi(s) \geq v_{\pi'}(s) \text{ for all } s \in \mathcal{S}. \tag{1.12}$$

There is always (at least) one policy that is better or equal to all the other policies and it is called *optimal policy*, denoted with π_*. All the optimal policies share the same *optimal state-value function* v_*, defined as

$$v_*(s) = \max_\pi v_\pi(s), \text{ for all } s \in (S). \tag{1.13}$$

They share as well the same *optimal action-value function*, denoted q_*, defined as

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{1.14}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. This can be rewritten in terms of v_* as:

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]. \tag{1.15}$$

Since v_* is a value function for a policy, it must satisfy the Bellman equation. The one that follows is known as *Bellman optimality equation*:

$$\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) && (1.16)\\
&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] && (1.17)\\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] && (1.18)\\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] && (1.19)\\
&= \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_*(s')]. && (1.20)
\end{aligned}$$

While the Bellman optimality equation for q_* is

$$q_*(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \Big| S_t = s, A_t = a] \tag{1.21}$$

$$= \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big]. \tag{1.22}$$

1.3 Dynamic Programming

Dynamic Programming (DP) is a general algorithmic framework that can be applied to those problems that possess *optimal substructure* and *overlapping subproblems*. In Reinforcement Learning DP is used to find optimal policies given a perfect model of the environment as in MDPs. As a matter of fact, MDPs present both characteristics needed, where Bellman equations allow recursive decomposition and the value function stores and uses subsolutions. Thus it is possible to apply dynamic programming to solve a MDP.

Although classical DP algorithms have limited utility, given their computational expense and the dependency on a perfect model, they are still important because they lie the foundations for many other algorithms.

Solving a MDP consists in - according to the *prediction* problem - evaluate the value function given a policy and - according to the *control* problem - in finding an optimal value function or optimal policy.

The prediction problem is solved by applying *policy evaluation*, by considering a sequence of approximate value functions v_0, v_1, v_2,... where the initial v_0 is chosen arbitrarily, where each following approximation is obtained applying the Bellman equation as update rule:

$$v_{k+1} = \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \tag{1.23}$$

$$= \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_k(s')\Big], \tag{1.24}$$

for all $s \in \mathcal{S}$.

This algorithm is called *iterative policy evaluation* and converges in general

to v_π as $k \to \infty$.

Policy evaluation is useful in order to perform *policy improvement*, namely finding a new policy that is better than the previous. For this task, it is possible to compute the action value function q for an action $a \neq \pi(s)$ and discover if it would be better to select a and follow $\pi(s)$ thereafter or to simply follow $\pi(s)$ in any state

$$q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \tag{1.25}$$

$$= \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_\pi(s')\Big]. \tag{1.26}$$

It is possible to extend this reasoning further to all possible actions and states, by acting *greedy* with the selection of the action a in state s that results in the best q value, to obtain a new policy π'

$$\pi'(s) = \arg\max_a q_\pi(s,a) \tag{1.27}$$

$$= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1} \mid S_t = s, A_t = a)] \tag{1.28}$$

$$= \arg\max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_\pi(s')]. \tag{1.29}$$

The combination of policy evaluation and policy improvement allows us to build a sequence of policies and value functions that improve monotonically:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} v_{\pi_2} \xrightarrow{I} ... \xrightarrow{E} \pi_* \xrightarrow{I} v_*, \tag{1.30}$$

where E and I above the arrows stand for *evaluation* and *improvement* respectively. Since finite MDPs have a finite number of policies, this process is granted to converge in a finite number of iterations.

To develop this approach further, by observing that it is not necessary to perform all the steps of policy iteration until infinity to obtain the best policy, we truncate the algorithm by using just one update of each state.

This new approach is called *value iteration*.

$$v_{k+1}(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \tag{1.31}$$

$$= \max_a \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma v_k(s') \Big], \tag{1.32}$$

for all $s \in \mathcal{S}$. The same result is obtained by noticing how the Bellman optimality equation can be turned into an update rule.
A major limitation in these DP approaches is that they require to perform operations over the whole state set, that can result in a greatly expensive task when the state is large. *Asynchronous* DP algorithms partially try to overcome this issue updating the values of the states in asynchronous fashion, in other words, some states are updated more times than others.

This interleaving process between policy evaluation and improvement, with all its variants, is referred to as *generalized policy iteration*. Almost all RL algorithms are described by this process where the policy is improved towards the value function and vice versa, as shown in Figure 1.3. This loop ends only when we reach a policy that is greedy with respect to its own value function, that implies optimality for both of them according to Bellman optimality equation.
Another way to see the process is in terms of two different goals, the two converging lines represented in Figure 1.4.

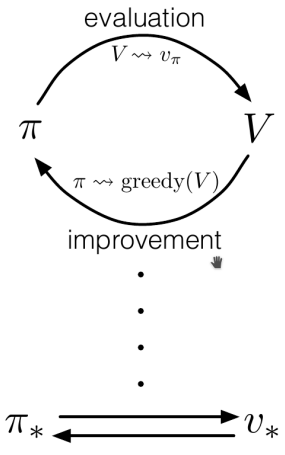


Figure 1.3: Generalized Policy Iteration (GPI) loop

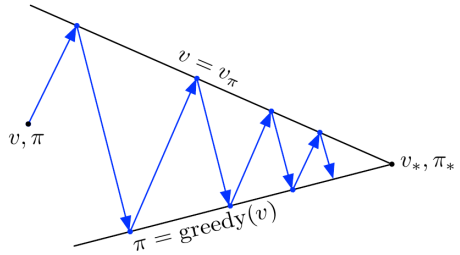


Figure 1.4: Alternative representation of GPI

1.4 Monte Carlo Methods

Monte Carlo methods are learning methods to estimate value functions and find optimal policies. Differently from DP, they do not require complete knowledge of the environment, but only *experience* intended as states, actions, and rewards sampled from the environment. These methods are based on averages of sample returns and are applied only to episodic tasks. The estimates of value function and policy are updated only at the end of an episode.

As for the prediction problem, Monte Carlo methods estimate the value function in a state, based on its definition of expected cumulative future discounted reward. One way to solve it, is to simply average over the returns obtained after visits to that state. After a number of visits, this average is bounded to converge to the expected return.

After defining a *visit* s as the occurrence of a state in one episode, Monte Carlo differentiates between a *first-visit* method and a *every-visit* method: in the former, v_π is estimated by averaging on the returns following first visits to s, while the latter averages the returns following all visits to s.

Similarly as the estimate of the state-value function, Monte Carlo estimate the action-value function $q_\pi(s, a)$ using the concept of visit, with the difference that visits here are to state-action pairs rather than to states. A state-action s, a pair is visited if a was chosen in a state s. The first-visit

method averages the returns that follow the first time in an episode that the action was picked in that state, while every-visit averages on all the returns that followed all the visits.

In the control problem, MC methods draw from the idea of generalized policy iteration (GPI) already presented, thus we consider again a modified version of classical policy iteration:

$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} q_{\pi_2} \xrightarrow{I} ... \xrightarrow{E} \pi_* \xrightarrow{I} q_*, \tag{1.33}$$

where E stands for policy evaluation and I for policy improvement. We suppose to observe an infinite number of episodes and that they are generated with *exploring starts*, that is, they all begin with state-action pairs randomly selected to cover all possibilities.
Policy evaluation is done in the same known manner, while policy improvement is achieved by making the policy greedy with respect to the current value function. The one that follows is the greedy policy:

$$\pi(s) = \arg\max_a q(s, a) \tag{1.34}$$

Subsequently, policy improvement is done by constructing each π_{k+1} as the greedy policy (as defined above) with respect to q_{π_k}. To avoid the first assumption, it is sufficient to alternate between evaluation and improvement on an episode-by-episode basis in the GPI.
To remove the second assumption on exploring starts or in other words, to ensure that all actions are selected, it is necessary to introduce the definitions of on-policy and off-policy methods.
On-policy methods attempt to evaluate or improve the policy that is used to make decisions while *off-policy methods* evaluate or improve a policy different from that used to generate the data. We can use both on-policy and off-policy to overcome this assumption.
In on-policy, the policy is *soft* meaning that $\pi(a \mid s) > 0$ for all $s \in S$ and all $a \in A(s)$, gradually shifted to a deterministic optimal policy. Without exploring starts, it is not enough to improve the policy by making it greedy

with respect to the current value function, because that would lead to lack of further exploration of non-greedy actions. However, in GPI it is important just to move the policy *towards* a greedy policy. For any ϵ-soft policy π, any ϵ-greedy policy with respect to q_π is guaranteed to be better than or equal to π.
The off-policy approach solves the dilemma of a policy trying to learn action values conditionally on optimal behaviour while still behaving non-optimally to allow exploration. This is done through decoupling of the policy in two: the policy being learned about called *target policy* and the policy used to generate the data, called *behavior policy*.
Almost all of these methods employ *importance sampling*, that is, weighting returns by the ratio of the probabilities of taking the observed actions under the two policies.

1.5 Temporal-Difference Learning

Temporal-difference (TD) learning is a combination of Monte Carlo and dynamic programming (DP) ideas. Like Monte Carlo, TD can learn without a precise knowledge of the environment, from experience, and like DP, it updates estimates using other learned estimates, using a mechanism called *bootstrapping*.
In the prediction problem, like Monte Carlo, given some experience following a policy π, TD learning updates its estimate V of v_π for nonterminal states S_t occurring in that experience. However, they don't need to reach the end of an episode, but they wait only until the next time step. In the simplest form of TD, known as $TD(0)$ or *one-step TD* the update rule is:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1} - V(S_t))\Big], \tag{1.35}$$

while in Monte Carlo the target for the update is the return G_t, here, the target is $R_{t+1} + \gamma V(S_{t+1}$ (known as *TD target*).
The quantity $R_{t+1} + \gamma V(S_{t+1} - V(S_t)$ is called *TD error*, since it measures a

difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$.

1.5.1 SARSA

SARSA (State-Action-Reward-State-Action) is an on-policy control method to find an optimal policy, defined by the following update rule [6]:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]. \tag{1.36}$$

Algorithm 1 SARSA

initialize Q arbitrarily, e.g. to 0 for all states, set action value for terminal states as 0
for *each episode* **do**
  initialize state s
  for *each step of episode, state s is not terminal* **do**
    $a \leftarrow$ action for s derived by Q, e.g. ϵ-greedy
    take action a, observe r, s'
    $a' \leftarrow$ action for s' derived by Q, e.g. ϵ-greedy
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$, $a \leftarrow a'$
  end for
end for

1.5.2 Q-Learning

Q-Learning is an off-policy TD control algorithm, defined by the rule:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)\Big] \tag{1.37}$$

where Q is the learned action-value function that directly approximates q_*, the optimal action-value function.
From the rule, follows this algorithm [6] :

Algorithm 2 Q-Learning

initialize Q arbitrarily, e.g. to 0 for all states, set action value for terminal states as 0
for *each episode* **do**
  initialize state s
  for *each step of episode, state s is not terminal* **do**
    $a \leftarrow$ action for s derived by Q, e.g. ϵ-greedy
    take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max'_a Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$
  end for
end for

1.6 Multi-Agent Reinforcement Learning

In a multi-agent system (MAS) multiple agents interact sharing the same environment [25]. In this domain, MDPs are generalized to *stochastic games* or *Markov games*.
Let us denote n number of the agents, S a discrete set of states and A_i, $i = 1, 2, ..., n$ a set of actions for each agent i. It is possible to define the joint action set for all agents as $A = A_1 \times A_2 \times ... \times A_n$. The state transition probability is $p : S \times A \times S \to [0, 1]$ and the reward function is $r : S \times A \times S \to \mathbb{R}^n$. The value function of each agent depends on the joint action and joint policy, which is characterized by $V^\pi : S \times A \to \mathbb{R}^n$.
MARL introduces a series of challenges to those already present in single agent RL, such as the *curse of dimensionality* that becomes even more problematic given the exponential growth of the state-action space, the problem of specifying a suitable goal, since agents' returns are correlated and cannot be maximized independently - from this the difficulty in shaping the reward, both in *cooperative* settings, where agents have a common goal, *competitive* and *mixed*.
In MARL, *non-stationarity* occurs because agents in a shared environment

potentially interact with each other and learn concurrently. This continuous interaction leads to a constant modification in the observed environment, and the Markov property does not hold anymore (also Q-Learning is not guaranteed to converge).

The exploration-exploitation dilemma is made more complex since agents need to explore not only to obtain more knowledge about the environment, but also on other agents. On the other hand, too much exploration can lead to destabilization of the other agents that are concurrently learning from the environment and the agent as well.

Chapter 2

Deep Q Networks (DQN)

In this chapter we explain how neural networks can be used in combination with the Q-Learning algorithm [21] [22] as function approximators in DQN [8], from the vanilla architecture to further describing some of the major advancements proposed in literature.

2.1 Neural Networks

Estimating Q values when action and state spaces are large can soon become an intractable problem: in this scenario, deep neural networks are a useful solution to approximate various components in a RL problem, such as policies $\pi(s, a)$ or values $q(s, a)$. The parameters of these networks are usually trained with gradient descent in order to minimize some loss function.
Neural networks are a mathematical model used for *function approximation*. The most simple model is the *deep feedforward network* or *multilayer perceptron* (MLP). A feedforward network defines a mapping $\boldsymbol{y} = f^*(\boldsymbol{x}; \boldsymbol{\theta})$ and learns the parameters $\boldsymbol{\theta}$ that grant the best approximation.
They are considered *networks* since they are composed of a sequence of functions, where each function is a *layer* in the network. The sequence goes from the first layer called *input layer* to the last, the output layer. The behavior

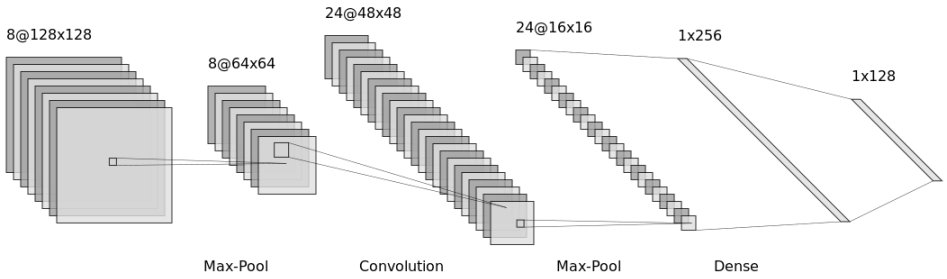


Figure 2.1: Example of CNN

of the middle layers is not specified by the training data, so they are known as *hidden layers*. The output values at these layers are chosen by *activation functions*.
Common networks that are used in Deep Reinforcement Learning and have been used especially in the first successful approaches applied to video games, are *convolutional neural networks (CNNs)*.
CNNs are a specialized kind of neural network for processing data that has grid-like topology, such as images. They employ *convolution*, that is a specialized kind of linear operation, in place of simple matrix multiplication in at least one of their layers.

2.2 Deep Q-Learning

DQN was introduced in [7] [8] as first algorithm that successfully combined deep neural networks and reinforcement learning. In DQN a convolutional neural network was trained to play a range of Atari 2600 games, using as input data the raw pixels from the images, a high-dimensional visual input, with the aim of reaching performances comparable to humans.

DQN makes use of two different techniques to enable relatively stable learning: *experience replay* and *target networks*. At each time step t after

Figure 2.2: Screen shots from five Atari 2600 Games: (*Left-to-right*) Pong, Breakout, Space Invaders, Seaquest, Beam Rider

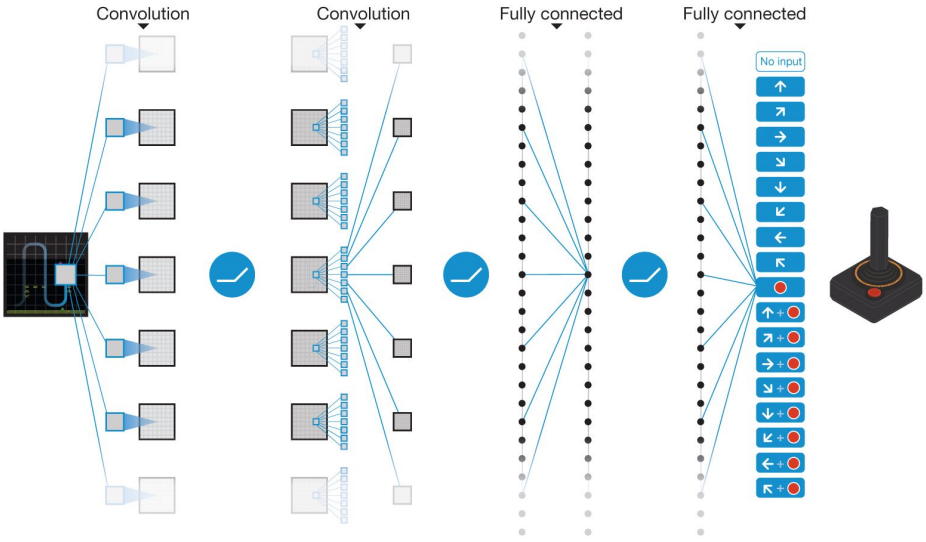


Figure 2.3: DQN network architecture

selecting an action ϵ-greedily with respect to the action values, a tuple of experience $(S_t, A_t, R_{t+1}, \gamma_{t+1}, S_{t+1})$ is saved into a replay memory buffer that can store up to one million transitions.

The weights of the network are optimized using stochastic gradient descent using as loss to minimize:

$$(R_{t+1} + \gamma_{t+1} \max_{a'} q_{\bar{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t))^2 \tag{2.1}$$

where t refers to the time step of a transition randomly chosen from the replay buffer.

Then, back-propagation through gradient descent is made only into the parameters θ of the *online network*, that is the network used to pick actions, whereas the *target network* with parameters $\bar{\theta}$ is updated only after a certain

number of timesteps as copy of the online network, and it is not directly optimized.

Algorithm 3 Deep Q-Learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode = 1, M **do**
  Initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max'_a Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

DQN showed successful results, producing for some games even better scores than the ones obtained by professional players.

Table 2.1 Comparison of games scores between DQN with methods from literature and a professional human tester

Game	Random Play	Best Linear Learner	Contingency (SARSA)	Human	DQN (± std)	Normalized DQN (% Human)
Alien	227.8	939.2	103.2	6875	3069 (±1093)	42.7%
Amidar	5.8	103.4	183.6	1676	739.5 (±3024)	43.9%
Assault	22.4	628	537	1496	3359 (±775)	246.2%
Asterix	210	987.3	1332	8503	6012 (±1744)	70.0%
Asteroids	719.1	907.3	89	13157	1629 (±542)	7.3%
Atlantis	12850	62687	852.9	29028	85641 (±17600)	449.9%
Bank Heist	14.2	190.8	67.4	734.4	429.7 (±650)	57.7%
Battle Zone	2360	15820	16.2	37800	26300 (±7725)	67.6%
Beam Rider	363.9	929.4	1743	5775	6846 (±1619)	119.8%
Bowling	23.1	43.9	36.4	154.8	42.4 (±88)	14.7%
Boxing	0.1	44	9.8	4.3	71.8 (±8.4)	1707.9%
Breakout	1.7	5.2	6.1	31.8	401.2 (±26.9)	1327.2%
Centipede	2091	8803	4647	11963	8309 (±5237)	63.0%
Chopper Command	811	1582	16.9	9882	6687 (±2916)	64.8%
Crazy Climber	10781	23411	149.8	35411	114103 (±22797)	419.5%
Demon Attack	152.1	520.5	0	3401	9711 (±2406)	294.2%
Double Dunk	-18.6	-13.1	-16	-15.5	-18.1 (±2.6)	17.1%
Enduro	0	129.1	159.4	309.6	301.8 (±24.6)	97.5%
Fishing Derby	-91.7	-89.5	-85.1	5.5	-0.8 (±19.0)	93.5%
Freeway	0	19.1	19.7	29.6	30.3 (±0.7)	102.4%
Frostbite	65.2	216.9	180.9	4335	328.3 (±250.5)	6.2%
Gopher	257.6	1288	2368	2321	8520 (±3279)	400.4%
Gravitar	173	387.7	429	2672	306.7 (±223.9)	5.3%
H.E.R.O.	1027	6459	7295	25763	19950 (±158)	76.5%
Ice Hockey	-11.2	-9.5	-3.2	0.9	-1.6 (±2.5)	79.3%
James Bond	29	202.8	354.1	406.7	576.7 (±175.5)	145.0%
Kangaroo	52	1622	8.8	3035	6740 (±2959)	224.2%
Krull	1598	3372	3341	2395	3805 (±1033)	277.0%
Kung-Fu Master	258.5	19544	29151	22736	23270 (±5955)	102.4%
Montezuma's Revenge	0	10.7	259	4367	0 (±0)	0.0%
Ms. Pacman	307.3	1692	1227	15693	2311 (±525)	13.0%
Name This Game	2292	2500	2247	4076	7257 (±547)	278.3%
Pong	-20.7	-19	-17.4	9.3	18.9 (±1.3)	132.0%
Private Eye	24.9	684.3	86	69571	1788 (±5473)	2.5%
Q*Bert	165.9	613.5	960.3	13455	10596 (±3294)	78.5%
River Raid	1339	1904	2650	13512	8316 (±1049)	57.3%
Road Runner	11.5	67.7	89.1	7845	18257 (±4268)	232.9%
Robotank	2.2	28.7	12.4	11.9	51.6 (±4.7)	509.0%
Seaquest	68.4	664.8	675.5	20182	5286 (±1310)	25.9%
Space Invaders	148	250.1	267.9	1652	1976 (±893)	121.5%
Star Gunner	664	1070	9.4	10250	57997 (±3152)	598.1%
Tennis	-23.8	-0.1	0	-8.9	-2.5 (±1.9)	143.2%
Time Pilot	3568	3741	24.9	5925	5947 (±1600)	100.9%
Tutankham	11.4	114.3	98.2	167.6	186.7 (±41.9)	112.2%
Up and Down	533.4	3533	2449	9082	8456 (±3162)	92.7%
Venture	0	66	0.6	1188	3800 (±238.6)	32.0%
Video Pinball	16257	16871	19761	17298	42684 (±16287)	2539.4%
Wizard of Wor	563.5	1981	36.9	4757	3393 (±2019)	67.5 %
Zaxxon	32.5	3365	21.4	9173	4977 (±1235)	54.1%

In the taxonomy of RL algorithms (see Figure 2.4) DQN belongs to the *model-free* subtree.
The difference between the two branches is that, in *model-based* the agent has access to a model of the environment, that is a function which predicts state transitions and rewards, while in model-free, the agent has to learn a model of it. The main advantage of having a model is that the agent can think ahead, already knowing what would happen for a range of possible actions, while deciding for the best option. The results from this planning can later be translated into a policy. The problem is that often, a ground-truth model of the environment is not available, forcing the agent to learn it from experience.

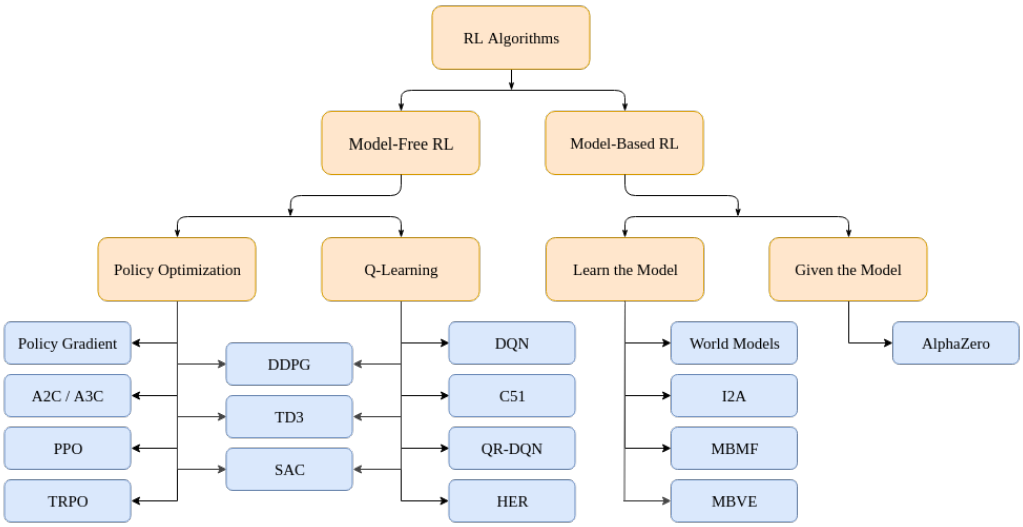


Figure 2.4: Non-exhaustive taxonomy of RL algorithms from OpenAI [26]

In the following subsections we discuss some useful modifications presented in literature that applied to the standard DQN algorithm here presented can lead to significant improvements in performance.

2.2.1 Double Q-Learning

In order to overcome the overestimation bias that affects standard Q-Learning due to the the maximization step in Equation 2.1, Double Q-Learning [30] decouples the selection of the action from its evaluation employing two independent models. It was then shown in [29] that with the use of an *online network* and a *target network* it can be successfully combined with DQN, by using the following loss function:

$$(R_{t+1} + \gamma_{t+1} q_{\bar{\theta}}(S_{t+1}, \arg\max_{a'} q_\theta(S_{t+1}, a')) - q_\theta(S_t, A_t))^2 \tag{2.2}$$

where $\bar{\theta}$ are the weights of the target network, and θ are the weights of the online network. The greedy policy is evaluated according to the online network, but using the target to estimate its value.

2.2.2 Prioritized Replay

In the standard experience replay buffer, transitions are sampled randomly, regardless of their significance, although ideally it would be better to sample more frequently those transitions from which there is more to learn. With prioritized replay buffer [30] transitions are sampled with probability p_t relative to the last encountered absolute *TD error*, as a measure of the expected learning progress:

$$p_t \propto \mid R_{t+1} + \gamma_{t+1} \max_{a'} q_\theta(S_{t+1}, a') - q_\theta(S_t, A_t) \mid^\omega, \tag{2.3}$$

where ω is a hyper-parameter that determines the shape of the distribution.

2.2.3 Dueling networks

The dueling network [31] [32] is a type of network architecture that uses two streams of computation, a *value* stream, for the state-value function and an *advantage* stream for the state-dependent action advantage function, sharing a convolutional encoder, and combined by a special aggregator layer

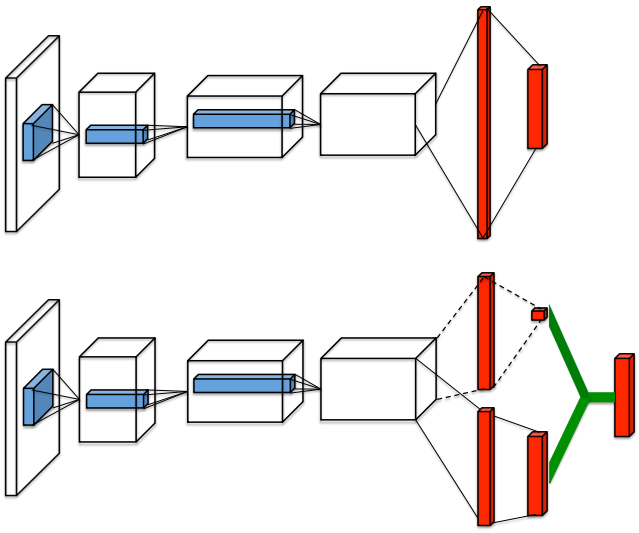


Figure 2.5: Comparing classical DQN and dueling network architecture

to produce an estimate of the state-action value function Q. The combination of the two streams corresponds to a specific factorization of the action values:

$$q_\theta(s,a) = v_\eta(f_\xi(s)) + a_\psi(f_\xi(s),a) - \frac{\sum_{a'} a_\psi(f_\xi(s),a')}{N_{actions}}, \tag{2.4}$$

where ξ, η, and ψ are, respectively, the parameters of the shared encoder f_ξ, of the value stream v_η and the advantage stream a_ψ; and $\theta = \xi, \eta, \psi$ is their concatenation.

The main benefit of this architecture is the ability of learning which states are or are not valuable without having to learn the effect of each action for each state.

2.2.4 Multi-step learning

Instead of accumulating one single reward and then using the greedy action at the next step to bootstrap as in simple Q-Learning, *multi-step* targets can be used in alternative. Intuitively, the idea is to look ahead to the next n rewards, states and actions. It is first necessary to define the truncated n-step return from a given state S_t

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1}. \tag{2.5}$$

Then we can define an alternative loss for DQN, that utilizes these truncated returns:

$$(R_t^{(n)} + \gamma_t^{(n)} max_{a'} q_{\bar{\theta}}(S_{t+n}, a') - q_\theta(S_t, A_t))^2. \tag{2.6}$$

This mechanism can lead to faster learning when n is optimally tuned.

2.2.5 Distributional RL

This technique learns to approximate the distribution of returns, called Z, instead of the expected return, estimated by the Q action value function in Q-Learning [34]. In other words, Z is a mapping from state-action pairs to distributions over returns, called *value distribution*.
The key idea is that return distributions must satisfy a distributional variant of Bellman's equation.
In particular, Z is modeled using a discrete distribution, whose support called $\boldsymbol{z}$, namely the values the distribution can assume, is based on a finite set of *atoms*, defined as:

$$Z = \{z^i \mid z^i = v_{min} + (i-1)\frac{v_{max} - v_{min}}{N_{atoms} - 1}, i \in 1, ..., N_{atoms}\} \tag{2.7}$$

where N_{atoms} is the number of atoms, v_{min} and v_{max} are the minimum and maximum values of the distribution, and are used as parameters, where each combination of values for the parameters originates a different return distribution. The probabilities of the atoms are determined by a model $\theta : S \times A \to \mathbb{R}^n$ such that z_i is sampled with probability p_i:

$$p_i(s, a) = \frac{e^{\theta_i(s,a)}}{\sum_j e^{\theta_j(s,a)}}. \tag{2.8}$$

The objective is to learn θ to approximate the true distribution of returns Z.

We call $d_t = (\boldsymbol{z}, \boldsymbol{p}_\theta(S_t, A_t))$, based on this support vector and these probabilities, the distribution to approximate.
In such way, it is possible to define a distributional variant of Q-Learning, by constructing first a new support for the target distribution, and then minimizing the *Kullbeck-Leibler* divergence - intuitively, a distance metric - that quantifies the difference between the distribution d_t and the target distribution $d'_t \equiv (R_{t+1} + \gamma_{t+1}\boldsymbol{z}, \boldsymbol{p}_{\bar{\theta}}(S_{t+1}, \bar{a}^*_{t+1}))$,

$$D_{KL}(\Phi_{\boldsymbol{z}} d'_t || d_t), \tag{2.9}$$

where Φ_z is a L2-projection of the target distribution onto $\boldsymbol{z}$, and $\bar{a}^*_{t+1} = \arg\max_a q_{\bar{\theta}}(S_{t+1}, a)$ is the greedy action with respect to the mean action values $q_{\bar{\theta}}(S_{t+1}, a) = \boldsymbol{z}^T \boldsymbol{p}_\theta(S_{t+1}, a)$ in state S_{t+1}.
As in the non-distributional case, a frozen copy of the parameters $\bar{\theta}$ can be used to build the target distribution.
A neural network represents this distribution, using $N_{atoms} \times N_{actions}$ outputs. A *softmax* is then applied to ensure the normalization of the distribution.

2.2.6 Noisy Nets

Noisy Nets [35] are a more sophisticated technique than ϵ-greedy to solve the exploration-exploitation dilemma, particularly useful in environments where it is required to perform many actions before collecting the first reward. Noisy Nets are characterized by a noisy linear layer, combining a deterministic and a noisy stream:

$$\boldsymbol{y} = (\boldsymbol{b} + \boldsymbol{W}\boldsymbol{x}) + (\boldsymbol{b}_{noisy} \odot \boldsymbol{\epsilon}^b + (\boldsymbol{W}_{noisy} \odot \boldsymbol{\epsilon}^w)\boldsymbol{x}), \tag{2.10}$$

where ϵ^b and ϵ^w are random variables and $\odot$ is the element-wise product.
This layer is used in substitution of any standard linear layer $\boldsymbol{y} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{x}$. Over time, the network will learn how to ignore the noisy stream - preferring exploitation over exploration - but it will do so at different rates in different parts of the state space, essentially allowing a *state-conditional* exploration.

2.2.7 Rainbow

Rainbow [27] is an integrated agent that arises from combining all the modifications presented so far, reaching in general the best performance among all.
The 1-step distributional loss is replaced by a multi-step variant, the target distribution is then defined as $d_t^{(n)} = (R_t^{(n)} + \gamma_z^{(n)}\boldsymbol{z}, \boldsymbol{p}_{\bar{\theta}}(S_{t+n}, a^*_{t+n}))$.
The resulting loss is thus the multi-step distributional loss

$$D_{KL}(\Phi_{\boldsymbol{z}} d_t^{(n)} || d_t) \tag{2.11}$$

where $\Phi_{\boldsymbol{z}}$ is again the projection onto $\boldsymbol{z}$.
This loss, originated from the fusion of multi-step learning and the distributional perspective, is combined then with Double Q-Learning, so the action in S_{t+n} is selected according to the online network as the bootstrap action a^*_{t+n}, and it is evaluated against a target network.
Prioritized replay buffer is employed but the prioritization is done not through the TD error as in the standard version, but through the KL loss, that is, what the algorithm is trying to minimize

$$p_t \propto \left(D_{KL}(\Phi_{\boldsymbol{z}} d_t^{(n)} || d_t)\right)^{\omega}. \tag{2.12}$$

Rainbow uses a dueling network architecture, adapted for distributions of returns. The shared layer $f_\epsilon(s)$ is fed into a value stream v_η with N_{atoms} outputs, and into an advantage stream a_ξ with $N_{atoms} \times N_{actions}$ outputs where $a^i_\xi(f_\xi(s), a)$ denotes the output corresponding to atom i and action a. For each atom, the two streams are combined and then passed to a softmax for normalization

$$p^i_\theta(s,a) = \frac{exp(v^i_\eta(\phi)) + a^i_\psi(\phi, a) - \bar{a}^i_\psi(s)}{\sum_j exp(v^j_\eta(\phi)) + a^j_\psi(\phi, a) - \bar{a}^j_\psi(s)}, \tag{2.13}$$

where $\phi = f_\xi(s)$ and $\bar{a}^i_\psi(s) = \frac{1}{N_{actions}} \sum_{a'} a^i_\psi(\phi, a')$.
Lastly, all the linear layers are replaced with noisy layers.

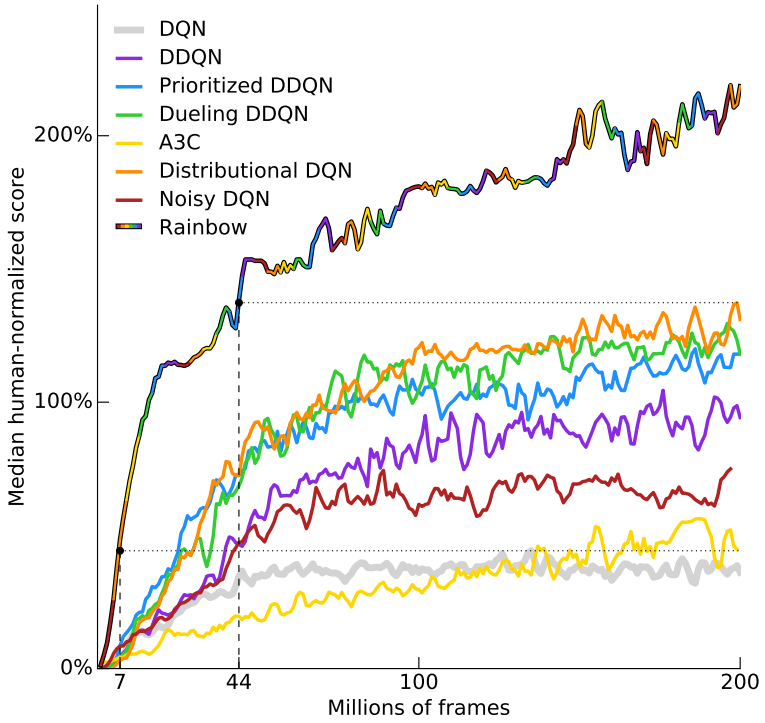


Figure 2.6: Performance of Rainbow compared to other DQN variants

Table 2.2 Preprocessing hyper-parameters in Rainbow

Hyper-parameter	Value
Grey-scaling	True
Observation down-sampling	(74, 74)
Frames stacked	4
Action repetitions	4
Reward clipping	[-1,1]
Terminal on loss of life	True
Max frames per episode	108K

Table 2.3 Additional hyper-parameters in Rainbow

Hyper-parameter	Value
Q network: channels	32, 64 x 64
Q network: filter size	8 x 8, 4 x 4, 3 x 3
Q network: stride	4, 2, 1
Q network: hidden units	512
Q network: output units	Number of actions
Discount factor	0.99
Memory size	1M transitions
Replay period	Every 4 agent steps
Minibatch size	32

Chapter 3

FLATLAND

Flatland is a discrete time multi-agent simulation of a railway environment . The Flatland environment was developed by AIcrowd [37] in collaboration with the Swiss Federal Railways (SBB) to foster progress and research in multi agent reinforcement learning for any rescheduling problem. This research could lead to important improvements in modern traffic management systems (TMS) in general, that are present not only in railway systems but also in other areas of transportation and logistics [36].

3.1 Background

The implementation of a simulation to face the problem is highly convenient to measure changes and try new methodologies before applying them to the real scenario. The Flatland simulation offers a ready-to-use mean to investigate approaches for automated traffic management systems (TMS), whose role is to select routes for all trains and decide on their priorities at switches in order to optimize traffic flow across the network. At the core of this simulation lies the general vehicle re-scheduling problem (VRSP), that states:

The vehicle rescheduling problem (VRSP) arises when a previously as-

signed trip is disrupted. A traffic accident, a medical emergency, or a breakdown of a vehicle are examples of possible disruptions that demand the rescheduling of vehicle trips [19].

3.2 Environment

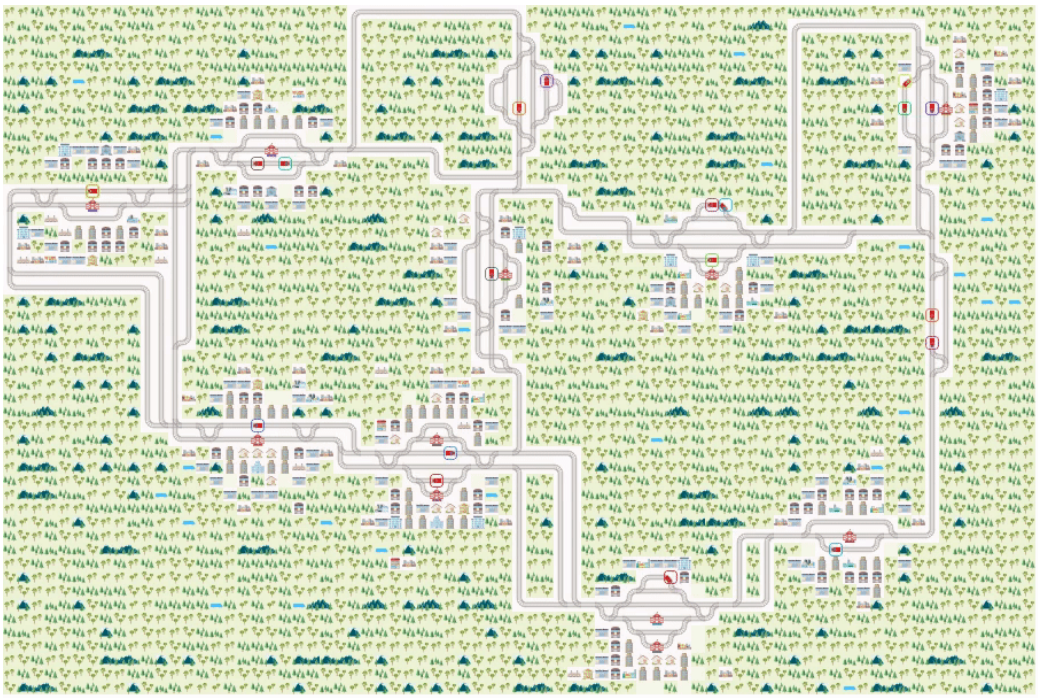


Figure 3.1: A representation of the Flatland environment

The simulation is represented onto a 2D grid environment, where the grid consists of cells, and restricted transitions between neighboring cells represent the railways. A cell is the elementary component of the grid, and has capacity one, that means, only one agent at a time can occupy that cell. An agent (train), has the ability to move in the grid, occupying with the passing of time, different cells according to the lecit transitions. The transitions allowed depend on the cell type and the agent orientation (North, East, South, West). There are 8 different cell types, according to their function within the grid:

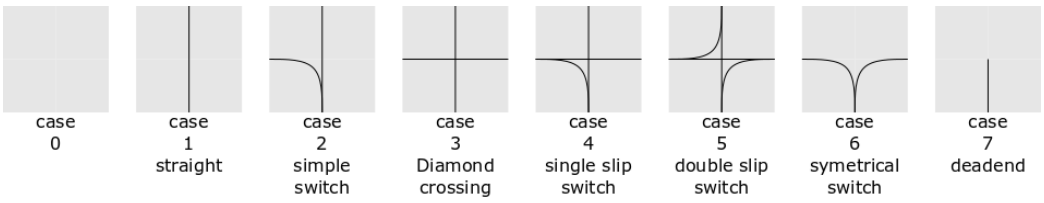


Figure 3.2: Types of cells available in Flatland

1. *Empty cell*: rendered in the environment as a building in a city or a green space. It can't be occupied by any agents;

2. *Straight rail*: navigation is possible only moving forward, there is no navigation choice;

3. *Simple switch*: an agent coming from South can choose to proceed forward or to take a turn left;

4. *Diamond crossing*: equivalent to two straight rails crossing each other, there is no possibility to turn, but the chance to conflict with some other agents;

5. *Single slip switch*: a diamond crossing with a possibility to turn from one of the directions available;

6. *Double slip switch*: as above, but with two possible directions from which to take a turn;

7. *Symmetrical switch*: an agent coming from South has the obligation to turn left or right;

8. *Dead-end*: an agent here can only stop or be forced to move backwards.

Each of these cells can appear rotated by 90°, 180°, 270°, creating other combinations of transitions.

3.2.1 Agent

An agent is a *train* on the map, starting from an initial position, with the goal of reaching a target *station*. The following features characterize an agent:

- *initial position*, cell that acts as starting point, indicated as a tuple of coordinates in the grid
- *initial direction*, (North, East, South, West)
- *direction*
- *target*, coordinates of the cell where the target station lies
- *moving*, if moving or stopped
- *speed data*
- *malfunction data*
- *handle*, the agent id
- *status*
- *position*, cell
- *old direction*, direction at the previous time step
- *old position*, position at the previous time step.

Speed

Being Flatland a simulation that aims to mock a realistic and mixed railway systems, multiple speed profiles are supported. The different speeds belong to different types of trains, such as fast passenger trains, normal passenger trains and freight trains. Speeds are specified using a float value, included in the range [0, 1] where 1 is the fastest profile and 0 the slowest.

For example, a value of 0.2 would indicate a train that is five times slower than the fastest. Agents keep their speed profile unaltered for the whole episode.

Malfunctions

Malfunctions occur in the environment as stochastic events. The aim is to simulate real case scenarios where the initial trains plans need to be rescheduled during operations due to minor occurrences of events such as delayed departures from train stations, malfunctions on trains or infrastructure or the weather.
Malfunctions are decided by a Poisson process that simulate delays and stops agents at random times for random durations.

Status

One agent during one episode can change its status, following four different combinations:

- *ready to depart*, the agent is not present in the environment, but it's ready to appear at its specified initial position
- *active*, the agent appears on the grid
- *done*, the agent has reached its target
- *done removed*, the agent was removed from the environment after reaching its target.

Action space

Despite the differences between agents, they all share the same action space, made of five different actions.

1. *Do Nothing*: If the agent is already moving it continues to move, if it is stopped it stays stopped;

2. *Deviate Left*: If the agent is at a switch with a possible left transition, it will turn left. If no turns are possible, the action has no effect and the agent keeps moving forward. If the agent is stopped, this action will start agent movement again if allowed by the transitions.

3. *Move Forward*: If stopped, the agent will start motion again. This will move the agent forward and choose the go straight direction at switches.

4. *Deviate Right*: Same as deviate left but for right turns.

5. *Stop*: This action causes the agent to stop. If the agent is stopped, it has no effect.

3.2.2 Tasks

In the Flatland environment, many tasks can be considered for the agent (or multiple agents) to perform.

Navigation

Figure 3.3: Agent navigating to target (blue path)

The most simple task is the navigation problem in a single-agent setting. In this scenario the train must reach a target position (*station*) from a starting point randomly generated in the map, in the least possible time.

Conflicts avoidance

In this task, multiple agents are present in the environment and in addition to the navigation task, complexity increases since orienting multiple trains in a shared map could lead to conflicts that must be predicted and avoided. In this task too, the goal is to schedule all the agents in a way that allows all of them to reach their targets as fast as possible.

Re-scheduling

In this scenario, planning ahead the routes for all the agents to navigate them to the target is not sufficient: due to failures, malfunctions and other disrupting accidents, some agents could stop functioning, blocking paths and forcing the others to re-plan their routes in order to reach their goals.

3.3 Observations

In the reinforcement learning problem, an agent receives at every time step *observations* from the environment, that represent somehow its state, and acts upon those to maximize a reward. Thus, in a model-free environment, shaping the observations is a key problem to solve for the agents to behave optimally. In the Flatland environment, some stock observations classes were developed as baselines for the problem, in the next subsections we discuss how they work.

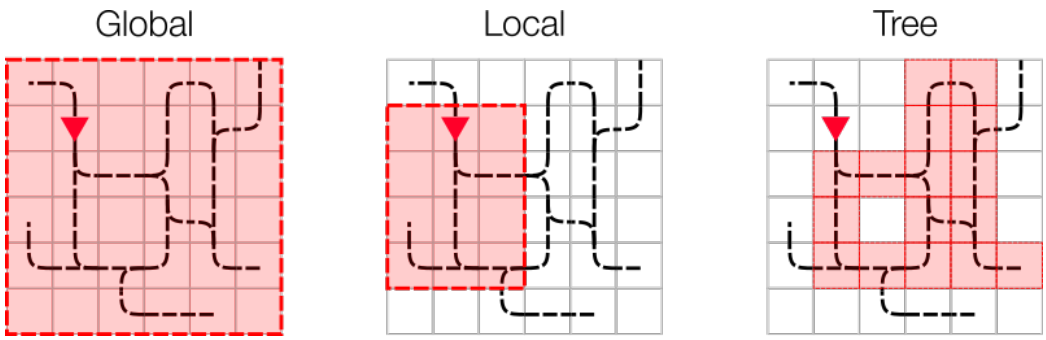


Figure 3.4: Global, local and tree observations in Flatland

3.3.1 Global Observations

As their name suggest, these observations give a global perspective on the environment, providing information about cell transitions, targets positions and agents positions and directions.
In particular the observation is composed of the following elements:

- transition map array with dimensions (env height, env width, 16), assuming 16 bits encoding of transitions.
- a multidimensional array of shape (env height, env width, 5) with
  - first channel containing the agents position and direction
  - second channel containing the other agents positions and directions
  - third channel containing agent/other agents malfunctions
  - fourth channel containing agent/other agents fractional speeds
  - fifth channel containing number of other agents ready to depart
- a multidimensional array of shape (env height, env width, 2) containing respectively the position of the given agent target and the positions of the other agents targets.

3.3.2 Local Observations

These observations work similarly as the global ones, but the view is limited to a local grid of dimensions *view height* $\times$ *view semiwidth*+1 around the agent. The observation is composed of the following elements:

- transition map array with dimensions
  (view height, 2 * view semiwidth + 1, 16), assuming 16 bits encoding of transitions (one-hot encoding)
- a multidimensional array of shape
  (view height, 2 * view semiwidth + 1, 5) with

  - first channel containing the agent position and direction (int on grid)
  - second channel containing active agents positions and directions (int on grid)
  - third channel containing agent/other agents malfunctions (int, duration)
  - fourth channel containing agent/other agents fractional speeds (float)
  - fifth channel containing directions of agents ready to depart (flag in correspondence to initial positions)

- a multidimensional array of shape (view height, 2 * view semiwidth + 1, 2)
  containing respectively the position of the given agent target/subtarget and the positions of the other agents targets/subtargets as one-hot encoding.

Essentially, the parameters *view height* and *view semiwidth* define what the agent 'sees' at each side. The base field view as a rectangle is defined with the agent facing North, and the origin lies at the upper-left corner. An *offset* parameter is used to move the agent along the height axis of this rectangle, from a position where it has only observations in front, to a position where it has only observations behind.

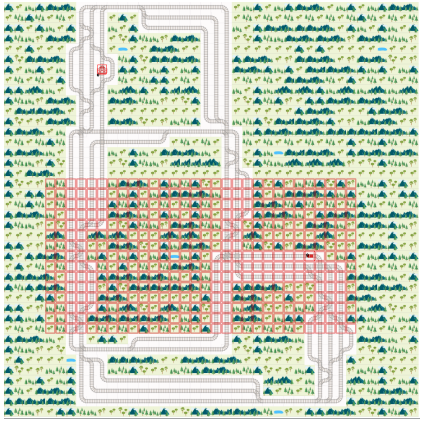


Figure 3.5: Local observations in the Flatland environment

3.3.3 Tree Observations

These observations are built exploiting the graph structure of the railway network.

In particular, observations are built only along allowed transitions on this graph, starting from the current position of the agent. As the agent moves along these transitions, a tree is built up, where a new node is created at every cell where the agent has different possibilities (e.g. a switch), a deadend or when the target is reached. Every node works as a branching point, from which subtrees are built along the allowed transitions according to the agent's orientation and in the four possible directions: left, forward, right, backward.

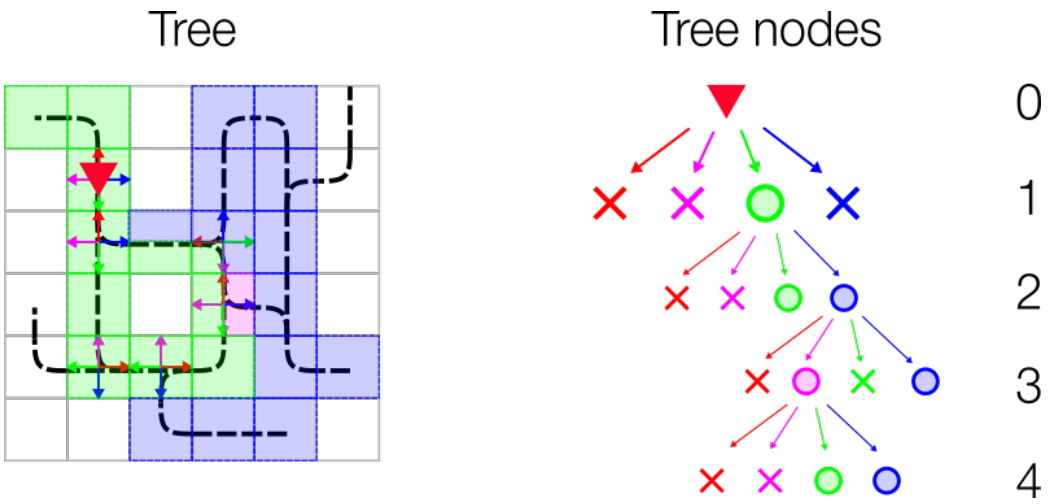


Figure 3.6: Extracting tree observation from the railway graph

The exploration starts from a root node (the agent's current position) to the leaves, following the branching directions, until a specified max depth is reached, stopping at terminal nodes and collecting information along the path.
The information gathered along a branch is stored in the tree as a node.

Node

A node is filled with the information collected along a path, consisting of the following features:

1. the distance in number of cells from target, if encountered;
2. the distance from the target of some other agent, if encountered;

3. the distance from another agent, if encountered;
4. the distance from a potential conflict with another agent, computed when two agents occupy the same cell at the same time;
5. the distance from a cell that is a switch, but can't be used by this agent;
6. the distance to the next branching point/node in the tree;
7. the minimum distance to the target, computed using Dijkstra shortest path algorithm;
8. the number of agents going the same direction;
9. the number of agents going opposite direction;
10. the number of malfunctioning agents;
11. the minimum speed of an agent encountered;
12. the number of agents ready to depart, namely, ready to enter the environment.

Data from each branch are arranged in a recursive manner inside a vector (*observations vector*), starting from the root node and then following the order *Left-Forward-Right-Back*.
Since nodes in the tree are computed for each of the four possibile directions, also along not allowed transitions, where placeholder values are used, the number of nodes at each level of the tree is fixed and depends on the current tree depth. Precisely, there are 4 elevated to the power of the level depth (starting from 0 for the root) for each level.
The total number of nodes in the tree is thus computed by summing the number of nodes at each level.
The observations vector, that contains the state representation, has a size of total number of nodes multiplied by the number of features stored in a node.

Chapter 4

Original work

In this chapter we explain and discuss some approaches that were tried to solve the tasks offered by the Flatland environment, starting from the most simple task of navigation, to the development of new classes of observations, *GraphObservations* and *RailObservations* to solve the conflicts problem in multi-agent settings, through the use of DQN and its more performing variants.
The code implementation of the described methodologies is available on GitHub at https://github.com/giulic3/flatland-challenge-marl
and https://github.com/misterdev/flatland-marl (refer to the latter for the Rail Observations approach).

4.1 Navigation task

The first task to be solved is the navigation problem in a single agent setting. In this simple scenario, an agent starting from a random position in a city in the map, must reach a station located a number of cells away. The goal for the agent is to reach the target in the minimum number of time steps.
Considering that traversing a cell costs a fixed number of time steps, that depends on the agent's speed, and that no other obstacles are present in

the environment, the problem of navigation translates into a shortest path problem on a graph, that is the underlying structure of the railway.
The algorithm used to compute the shortest path from the agent's current position (the *source*) and its target (the *destination*) is a variant of Dijkstra's algorithm [39] on graphs that have loops. From the current cell, the positions of the neighboring cells are found and every direction is walked through a *Breadth-First Search (BFS)* that keeps track of the nodes that were already visited and the distances walked to choose the minimum.
In this particular setting, the most straightforward way to shape rewards in order to make the agent reach its target fast, is to assign a negative reward (-1) for every time step spent navigating and giving a positive reward $(+1)$ when reaching the target.

To solve this task, the Tree Observations class was used and as model, a neural network with a dueling architecture was employed, where the convolutional layers were replaced by linear layers, starting from the input one, with a size in units equals to the observations vector size. A diagram of the architecture is represented in Figure 4.1.

We trained the network with a Double DQN with Randomized Replay Buffer, the approach showed successful results, allowing the agent to reach the target in the totality of the episodes, and showing a stable learning curve after a training of 6000 episodes. All the trainings in this round were completed using a cluster of CPUs Intel(R) Xeon(R) CPU E5420 @ 2.50GHz.

4.2 Avoid conflicts

In this second task of increased complexity, the experiments led showed the limitations of the basic observations classes, that according to our findings do not possess sufficient potential to solve the problem.
The Global Observations contain sparse information about the map, without providing the agent with specific data to follow to avoid the conflicts.

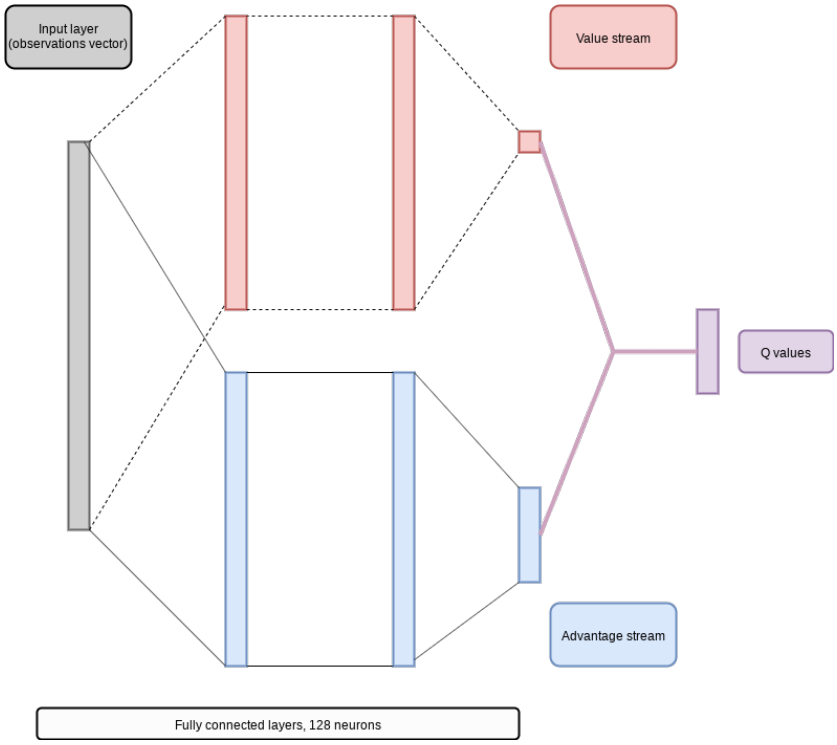


Figure 4.1: Neural network architecture

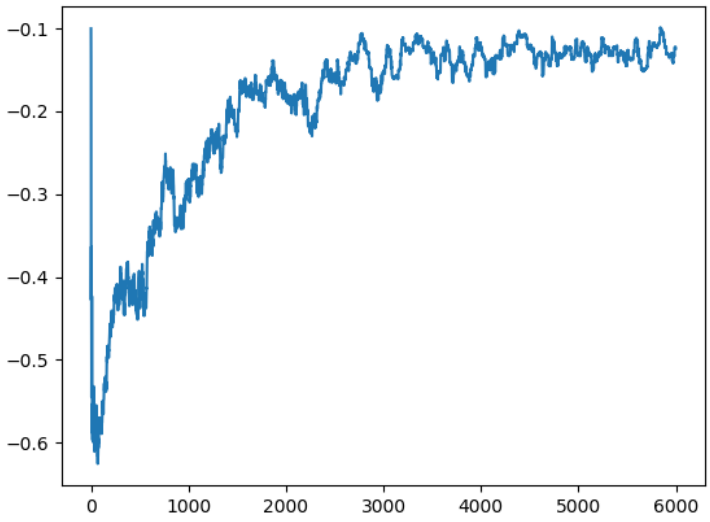


Figure 4.2: Learning curve in single agent navigation task

The Tree Observations are useful during navigation, since they exploit the natural structure of the railway as a graph, but are computationally expensive, where the effort is dependent on the chosen tree depth and the length of the prediction that is used to find the shortest path. This computation may be feasible in a single-agent setting, but becomes impractical in a multi-agent environment of 50 to 100 trains.
The Local Observations contain information only on a portion of the map, lacking any data on the agents' targets in case they lie outside of the local grid. A first approach to overcome this issue was attempted by adding subtargets into the grid. A *subtarget* defines an "exit point" in the cells window, that indicates the closest cell to the target in that window. However, the approach still gave scarce results.

After obtaining these findings, we decided to formulate the problem under a more simple perspective and thus implement new types of observations.

4.2.1 Graph Observations

The first idea was to decouple the problem in two levels: path selection or navigation (the *high* level) and conflicts avoidance the (*low* level). While the high level is solved by the shortest path algorithm (and potentially by other path finding techniques on graphs), the low level is approached by deep reinforcement learning methodologies.
In this view, since path selection is considered only at a high level, navigation choices like turns are not to be decided by the agent anymore, leading to a reduction of the action space to only two actions, *go* (0) and *stop* (1). The second one used only in necessary cases to avoid conflicts. The observations that are presented reflect this idea of the agent's acting only at the low level.

The new type of observations, called *Graph Observations* make use, as the name suggests, of the idea of the railway as a graph, as in Figure (4.3). The figure at the bottom is the abstract representation of the map at the top.

In this representation a *node* is a *switch* (or *fork* or *bifurcation point*) and an edge is a *path* between two nodes, also called *span* (of cells). Edges depend on connections entering or exiting a node. A node is seen as a global entity, connections are added from a global perspective, that is, without taking into account orientation of a particular agent. This means that in a multi-agent setting a switch for one agent, could not function as a switch for another, if the combination of node connections and agent current orientation does not allow that. The red node represents the target station and the green arrow is the agent.

Given that the path selection/navigation task is not solved by the DRL agent, our idea is to use the graph structure as a support in defining a new observation based on the path chosen, seen as a sequence of cells. The most straightforward data structure to frame this view is the array. Indeed, this abstract representation supports the construction of the observations as a vector of many concatenated layers: the agent has a sight on a limited number of cells that are lying in front of it in the path, and this observation can be enriched with many different features that analyze that path and can help the agent in conflicts avoidance. The parameter that is used to delimit the agent's view in the future here is referred to as *prediction depth* or *max prediction depth*.

Graph Observations are made of the following components, that will be later explained in details:

- occupancy info, multidimensional array of shape (max prediction depth, 2)
- forks, array of shape (max prediction depth)
- targets, array of shape (max prediction depth)
- priority and max priority encountered: two float
- number of malfunctioning agents, integer
- number of agents that are ready to depart, integer

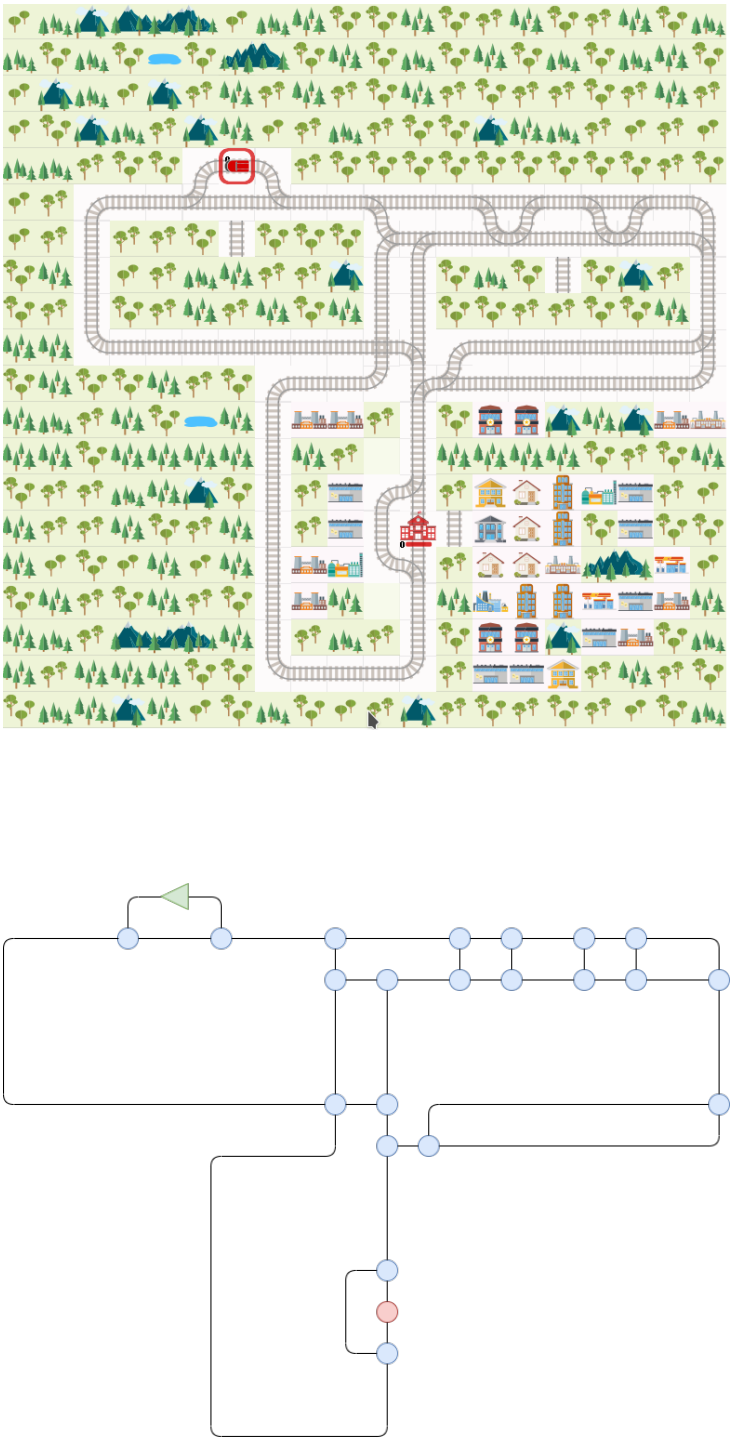


Figure 4.3: Example of graph extracted from a map

Occupancy information refers to data about agents' locations in the sequence of predicted cells. Since a cell in Flatland is a resource that can be occupied at each time step by one agent at maximum, it is important to define rules on how to determine the rights to occupy these cells, in order to avoid conflicting situations.

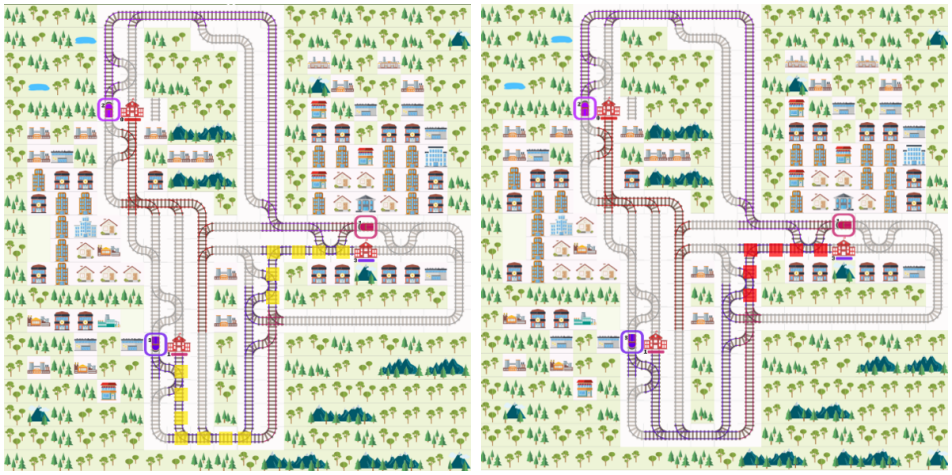


Figure 4.4: Example of conflict zones when comparing two agents' paths

We consider potential conflicts between agent 1 with target 1 and agent 3 with target 3.
In yellow, overlapping paths between the two, namely, sequences of cells that are in common in the prediction.
In red, a potential conflict zone, or overlapping of path in time (e.g. when agent 1 has speed lower than agent 3).

Starting from the sequences predicted for each agent, we compute paths that overlap in time, to check cells that will be occupied by two agents at the same time step (conflict). Occupancy information is then organized in two different layers: in the first one, possible conflict zones are computed - as stated above - as sequences of 1s marking the overlapping paths, while in the second layer, we deal with conflict zones that have already been occupied by some agent. In this case the agent on the conflict zone has *priority* on the span resource, and all the others must wait to enter the conflict zone, until

it is free again.
Forks are computed as the nodes in the graph representation, a cell in the map is eligible to fork if its group of available transitions matches a switch configuration.
Targets are represented by a one-hot encoding array, a target is a flag inside the observations vector that points to a cell that contains a target in the map.
Priority information consists in two values, one representing priority assigned to this agent and one representing the maximum priority encountered along the prediction sequence, if any another agent was encountered. The two values are used to estimate a comparison between two agents and establish a hierarchy in the episode. A notion of priority proved to be useful to overcome *deadlock* situations, when two agents predictions give rise to conflict zones but it is not clear which one should enter that area first and which would wait.
Priority is a key issue in our approach and many trials were made to define the most suitable one. In the end, we opted for a priority that is function of the agent status, its speed and its current distance to the target. (See the Appendix for the code in detail).
The *number of malfunctioning agents* and *agents that are ready to depart* are counters that are updated at each time step, since it is possible to have new malfunctions and new agents entering the environment at any step.

Rainbow for MARL

Graph observations as described in the previous section are then used in training in combination with a single-agent Rainbow implementation [40] adapted for a multi-agent system. The single prioritized replay buffer is replaced by multiple buffers, one assigned to each agent. Experiences are then stored relatively to the own replay buffer and during learning we sample a batch of experience tuples from one of the memories chosen randomly.

Results

In this section we collect results and findings obtained from the training on Graph Observations with Rainbow.

Table 4.1 Rainbow hyper-parameters in Flatland

Hyper-parameter	Value
Max number of steps	200
History length	1
Network hidden size	512
Std deviation of noisy layers	0.1
Atoms	51
Min value of distribution support	-10
Max value of distribution support	10
Replay memory capacity	10^5
Sampling frequency	4
Exponent in prioritized ER	0.5
Importance sampling weight in prioritized ER	0.4
Steps in multi-step return	3
Discount factor	0.95
Update steps of target network	10
Clip reward	False
Learning rate	0.0000625
Adam epsilon	1.5^{-4}
Batch size	32
Number of episodes before training	40

Hyper-parameters are taken from the standard Rainbow implementation, where we modified *history length*, *update target steps* and *learn start*. The *maximum number of steps* in Flatland depends on environment width, height and number of cities. The value here used is a threshold derived from observations and should be lower than the max expected from the environment (to speed up learning).

Table 4.2 Flatland environment base parameters

Parameter	Value
Width	20
Height	20
Number of agents	4
Max number of cities	3
Max number of rails between cities	3
Max number of rails in a city	3
Dispose cities in a grid	True
Malfunction rate	10000
Min malfunction duration	20
Max malfunction duration	50
Prediction depth	60

Training was performed in relatively small environments to test the performance of the approach and eventually check later its generalization ability on larger environments. *Malfunction rate* is on purpose very high to avoid malfunctions to occur in one episode and keep the task simple.

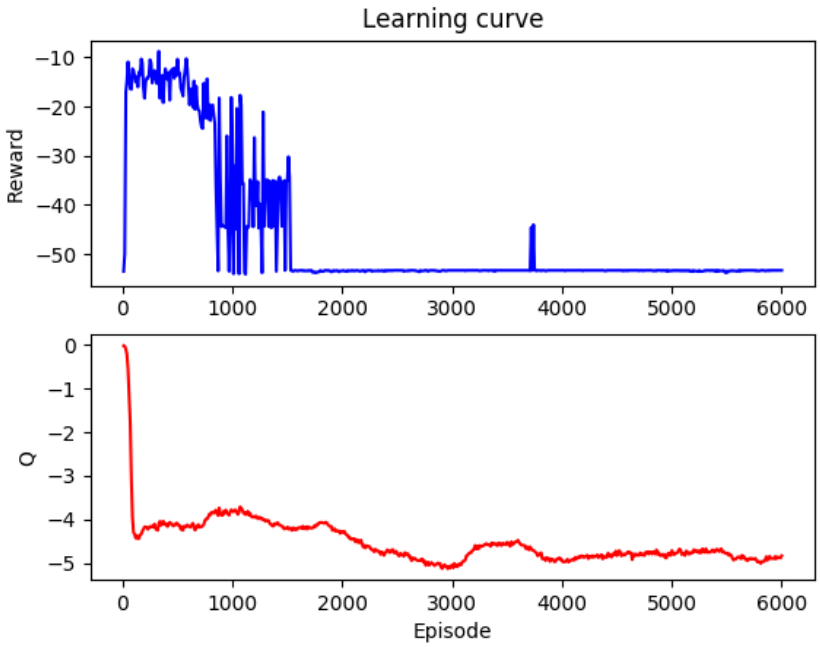
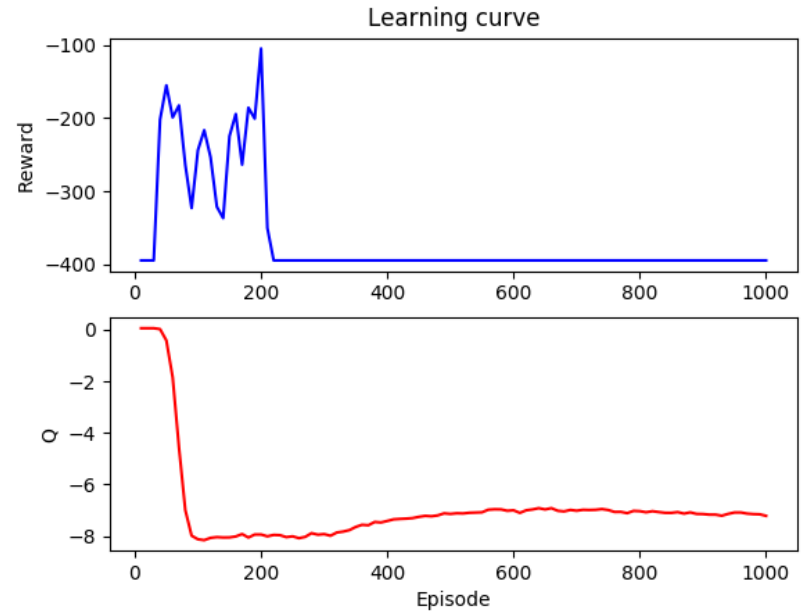
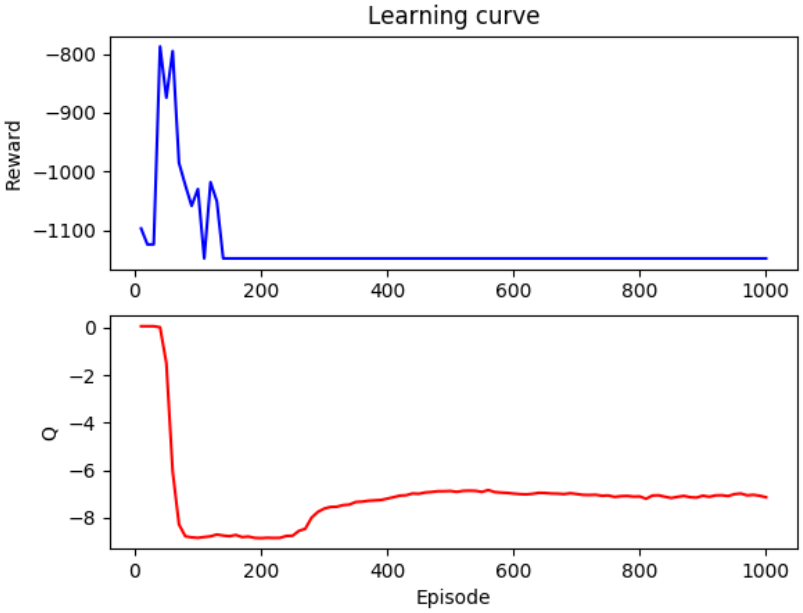


Figure 4.5: Learning curve with Rainbow in a single-agent setting

Training on 6000 episodes of 200 time steps each.
Evaluation is performed every 10 episodes during the learning process, averaging on a set of 10 episodes. We plot (in blue) the mean reward against the current episode number and (in red) the mean Q action value against the current episode number. From the charts we see how the learning diverges: initially the agents receive a reward correspondent to random actions, then the performance degenerates.



(a) Number of agents = 4, prediction depth = 60



(b) Number of agents = 10, prediction depth = 60

Figure 4.6: Comparing learning curves with different number of agents

Evaluation during the training is performed with the same parameters used in the single-agent setting. As in the other setting, the plots show how the agent is not able to learn using Graph Observations.

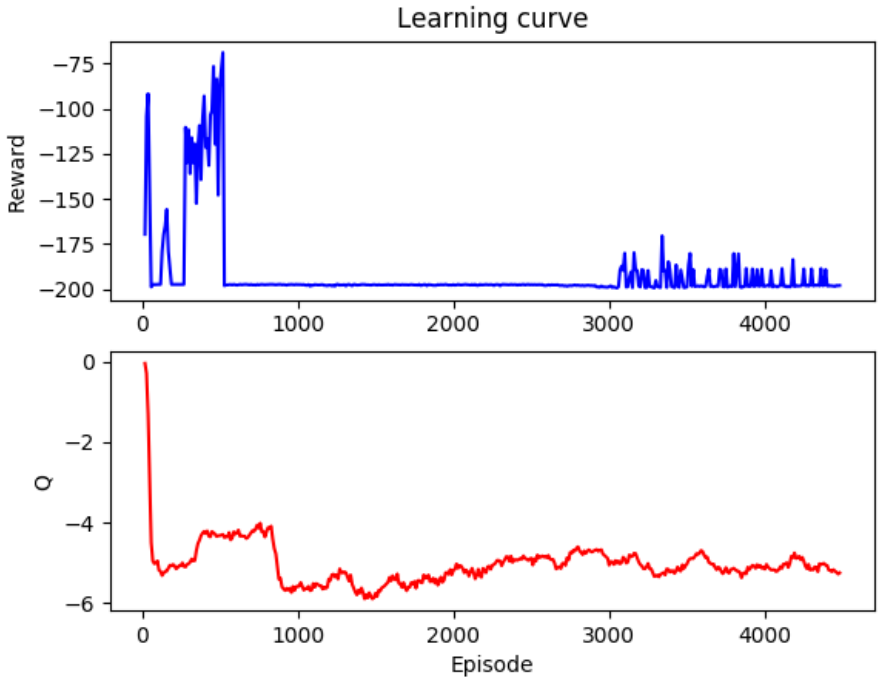


Figure 4.7: Learning curve with Rainbow in a multi-agent setting, number of episodes = 4000, prediction depth = 108

Here we raised the number of episodes of training to 4000, since Rainbow - as DQN in general - is not sample efficient, many experience tuples are usually needed for it to converge, this suggesting that increasing the number of episodes could lead to improvements in training. In addition, prediction depth is made longer and set to a more reasonable value for the size of the environment. As a matter of facts, given that agents can have different speed profiles, the slowest agent could not be able to see further enough if prediction depth is too low.

4.2.2 Rail Observations

The second observation class developed, called *Rail Observations*, continues to exploit the idea of occupancy of the agents on a graph structure, while considering the occupation of agents on edges of this graph, instead of occupation on single cells, summarizing the information into a bitmap.
In this graph the entity *node* is still a switch (or diamond crossing) and edges are the sequences of cells that connect nodes. In particular, nodes and edges are enumerated from the map and assigned a unique integer *id* to distinguish them. This enumeration starts as a process from the first node of the grid (cell [0,0]), then switches are identified by analyzing the possible transitions inside cells. Nodes are characterized by four entry (and exit) points, the cardinal directions North, East, South, West (N, E, S, W for brevity) that determine the possible connections between edges that are connected through that node. Information is stored into a *connections matrix*, a squared matrix where an element of indices $_{\text{ij}}$ is 1 if a connection from i to j is given and 0 otherwise, where $i, j \in N, E, S, W$ and all the matrices are then stored into a dictionary *connections* that maps node ids to their connections.

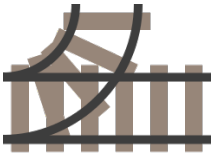


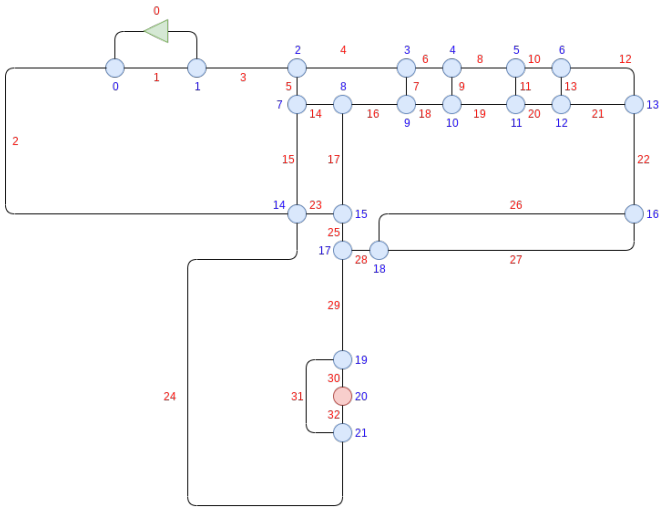
Figure 4.8: A switch with connections N-W and E-W

For example, the connections matrix associated to that switch is:

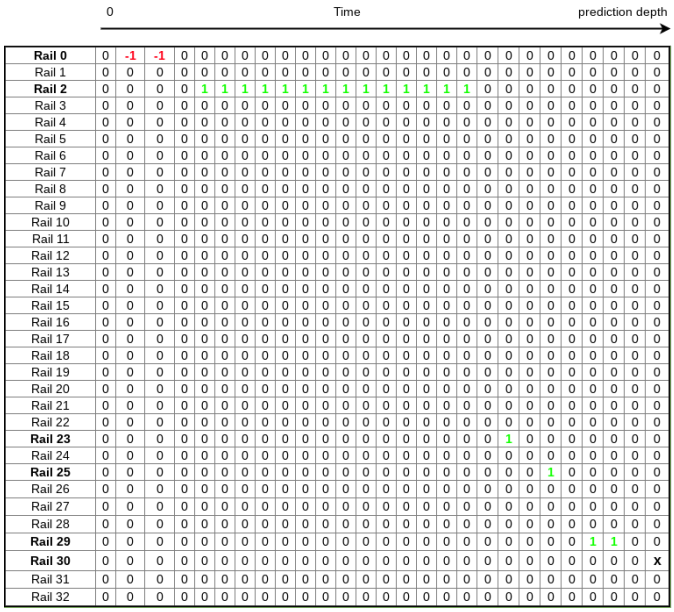
$$\begin{array}{c} \\ N \\ E \\ S \\ W \end{array} \begin{array}{c} \begin{array}{cccc} N & E & S & W \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Subsequently, from this dictionary edges are built through a visit of the nodes towards the possible directions (in order N-E-S-W) and information about them is stored into another dictionary called *info* that maps edge ids to a tuple of (CardinalNode1, CardinalNode2, edge length), where a CardinalNode is a named tuple representing a pair (node id, cardinal point).
The last important information that is stored in the process is the correspondence between edges and sequences of cells that compose them together with the traversal direction, which gives an ordering of the two cardinal nodes at the two extremes of the edge.

After defining the graph from the map it was necessary to find a suitable data representation for the input to be fed to the neural network for training. In order to do this we built for each agent a table, or *bitmap*, that expresses the current occupation of the train on the graph in time. In this representation, an edge or rail, is a resource that can be traversed in two different directions (+1 and -1), decided accordingly to the traversal direction defined when creating the edges.



(a) Graph extracted from a map, with nodes (switches) and edges (rails) labeled



	0	Time																							prediction depth
Rail 0	0	-1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 2	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
Rail 3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Rail 24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
Rail 26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
Rail 30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	x
Rail 31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rail 32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(b) Table (bitmap) filled from the graph representation

Figure 4.9: Rail Observations

Given the graph representation in (a), the agent (green arrow) can reach its target following the shortest path made of the sequence of nodes and edges: 0 - [2] - 2 - [14] - 23 - [15] - 25 - [17] - 29 - [19]- 30 - [20] (switches are in square brackets). A (+-1) in the bitmap indicates the presence of an agent on the row rail, where a sequence of bits indicates a time period of occupation - depending on the length of the edge and the agent's own speed, a train could traverse one cell in more than one time step.

The bitmap is a matrix of size (number of rails x maximum prediction depth +1), where rails are the edges (*rows* of the bitmap) and the maximum prediction depth refers to the maximum exploration limit of the shortest path predictor, expressing the notion of a path connected with time (*columns* of the bitmap). Essentially, the bitmap summarizes for one agent the occupation of rails in time steps during its path from source to target (see Figure 4.9).
An element ij in the bitmap can be:

- 0 : the agent is not occupying the rail i at time step j
- +1 : the agent occupies the rail i following the defined traversal direction at time step j
- -1 : the agent occupies the rail i with opposite direction with respect to the defined traversal direction at time step j.

In this approach, conflicts avoidance measures are implemented on data by analyzing the bitmaps of the other agents in the environment; in particular, to give the current agent indication about the presence and position of potentially conflicting agents, we build *heatmaps*, a positive one and a negative one, by summing up the other agents' bitmaps. The positive heatmap represents the occupation from rail traversed with positive direction (+1) and the negative heatmap express the presence of trains following the negative direction (-1), resulting each of them in a matrix that gives information on which are the rails on the map where conflicts are most probable to arise.

Table 4.3 Bitmaps for three different agents

	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9
rail 0	0	0	0	0	0	0	0	0	0	0
rail 1	0	0	0	0	0	0	0	0	0	0
rail 2	0	0	0	0	0	0	0	0	0	0
rail 3	0	+1	+1	+1	+1	0	0	0	0	0
rail 4	0	0	0	0	-1	-1	-1	-1	-1	-1

	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9
rail 0	0	+1	+1	+1	+1	0	0	0	0	0
rail 1	0	0	0	0	0	0	0	0	0	0
rail 2	0	0	0	0	0	-1	-1	0	0	0
rail 3	0	0	0	0	0	0	0	-1	-1	-1
rail 4	0	0	0	0	0	0	0	0	0	0

	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9
rail 0	0	+1	+1	+1	+1	0	0	0	0	0
rail 1	0	0	0	0	0	0	0	0	0	0
rail 2	0	0	0	0	0	0	0	0	0	0
rail 3	0	0	0	0	0	0	0	0	0	0
rail 4	0	0	0	0	0	0	0	0	0	0

Table 4.4 Positive and negative heatmaps

	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9
rail 0	0	2	2	2	2	0	0	0	0	0
rail 1	0	0	0	0	0	0	0	0	0	0
rail 2	0	0	0	0	0	0	0	0	0	0
rail 3	0	1	1	1	0	0	0	0	0	0
rail 4	0	0	0	0	0	0	0	0	0	0

	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9
rail 0	0	0	0	0	0	0	0	0	0	0
rail 1	0	0	0	0	0	0	0	0	0	0
rail 2	0	0	0	0	0	-1	-1	0	0	0
rail 3	0	0	0	0	0	0	0	-1	-1	-1
rail 4	0	0	0	0	-1	-1	-1	-1	-1	-1

As we did in the previous approach, we decided to simplify the problem reducing the action space to the two possible directions *go forward* or *stop*. However, in this framework, computation of alternative paths exiting from one rail is provided, and the network is forced to evaluate the best choice against the two actions for all the possible alternative bitmaps (called *altmaps*) that can be produced from these alternative paths. From the output Q values then, the action index corresponding to the max is picked, and a new path for the agent is set according to the altmap selected.

Thus, the RL agent chooses an action only in correspondence of nodes, in particular *before* switches, that are encoded in the bitmap, in the initial part of the following rail, as a sequence, or as a single. It can also choose actions during the initial phase, with status *ready to depart*, in order to enter the environment.
Inside the bitmap, the presence of a switch is represented by the end of a sequence of non-zero bits and the beginning at the following time step of a new sequence of bits at a different row.

Table 4.5 Switches on bitmaps are positioned at the end of non-zero sequences of bits.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
$rail_0$	0	+1	+1	+1	+1	0	0	0	0	0
$rail_1$	0	0	0	0	0	-1	-1	0	0	0
$rail_2$	0	0	0	0	0	0	0	0	0	0
$rail_3$	0	0	0	0	0	0	0	-1	-1	-1

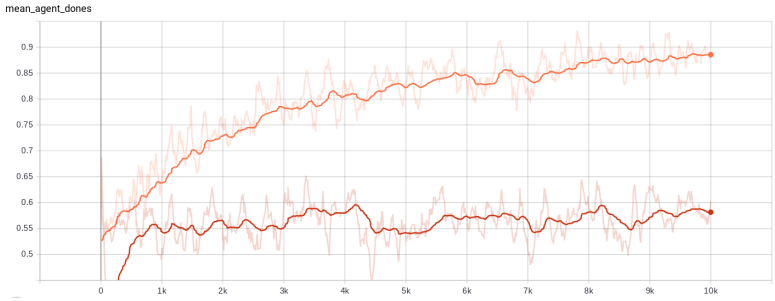
The final input for the network is thus made of bitmaps (of the current or of one alternative path) and two heatmaps concatenated, all preprocessed

with a padding in the matrix that considers the maximum number of rails that the map could have. In this way, the network is able to learn from data originating from different maps and episodes. The network determines the output action for each agent singularly.

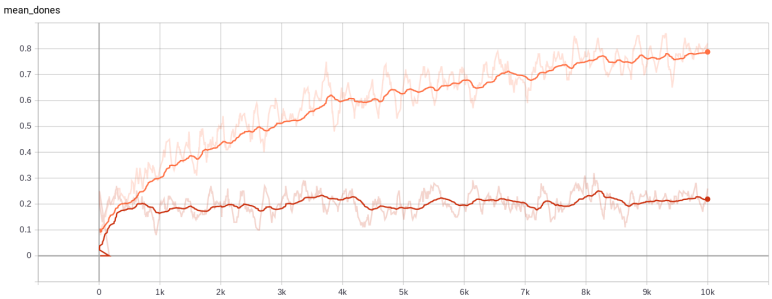
Results

The agent was trained on a cluster of machines equipped with GPUs NVIDIA GeForce GTX 1080, since as opposed to the Graph Observations approach, we observed how training with bitmaps could benefit from using GPUs instead of CPUs. The parameters used for the Flatland environment reflect the ones used in the Graph Observations approach, where the prediction depth was increased to 150 by default. We simplified the task by considering only agent with constant unitary speed and avoiding the occurrence of stochastic malfunctions. These assumptions, together with the use of small environments could be a good starting point to show the potential of the approach.
The the algorithm used for the training is a Double DQN with randomized replay buffer and ϵ-greedy exploration with the network having the same architecture as the previous approach, where the first layer was converted from linear to convolutional to process the bitmap structure.

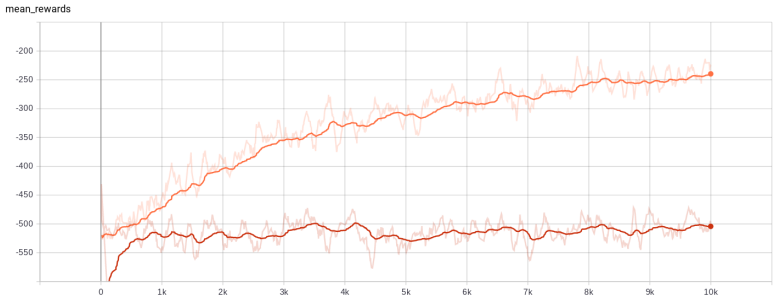
We report some charts obtained by plotting the relevant scalars using TensorBoard for PyTorch.



(a) Fraction of agents that reached their target

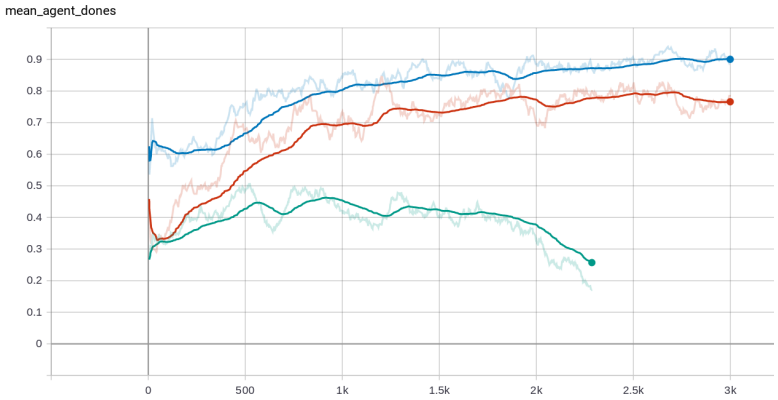


(b) Fraction of completed environments (100% of agents reached the target)

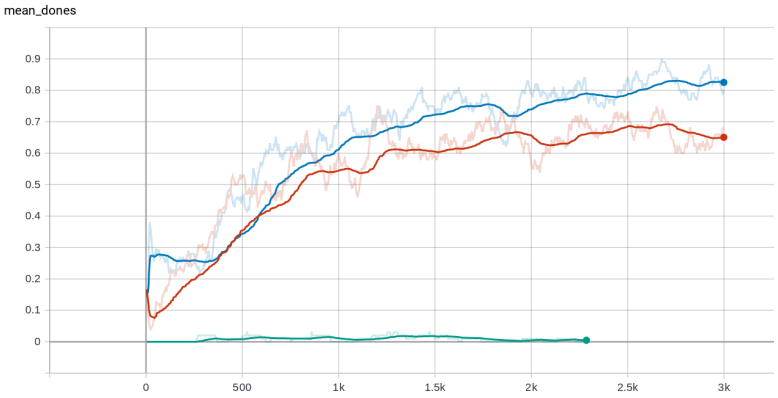


(c) Cumulative reward through episodes

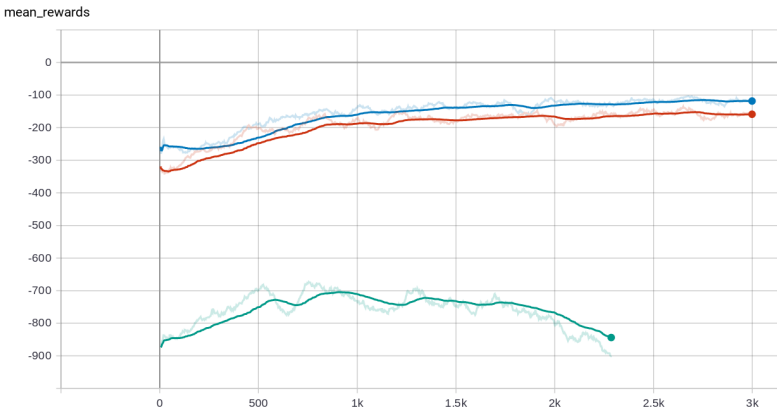
Figure 4.10: Comparison of performance between a learning agent (orange) and a random agent (red), the latter used as a baseline



(a) Fraction of agents that reached their target



(b) Fraction of completed environments (100% of agents reached the target)



(c) Cumulative reward through episodes

Figure 4.11: Other training experiments: on a bigger environment of 30x30 with 10 agents (green), reordering the rails in the bitmap (blue), reordering the rails and cutting the ones that are not traversed (orange)

Chapter 5

Discussion

As the plots show, results of using Graph Observations with Rainbow lead to a divergence of Q values during learning.
This behaviour is very common, DQN is in general hard to train and tune, and very slow to converge even with a successful state representation. The difficulties in finding convergence could be also an effect of the stochasticity of the environment we are trying to learn from, since the maps in the environment change at every episode.
According to our findings, after repeating the experiments with different hyperparameters and still obtaining not sufficient results, we believe that the approach we used could have been too simplistic to tackle the problem effectively. For example, the determination of the priorities between agents can be made more complex by considering also the network structure. In addition, path selection through the shortest path algorithm for single agent is probably too reductive, given that it does not take into account the presence of other agents.

As for the Rail Observations approach, the learning curves resulting from the experiments gave positive outcome, showing a constant improvement over the baseline in the performance of the agents until reaching a stable 90% of agents *done*. These good results show that the representation of the input data as a bitmap is a successful mechanism for the network to

learn. Further experiments would be trying the effectiveness of this approach on bigger environments, restore the fractionary speed and conceive more sophisticated bitmaps, and apply malfunctions with a different strategy for path re-calculation.

In the following, we point in detail the problems posed by this challenge, the limitations of the approaches used and conceivable related efforts that could result in improvements in performance.

DQN and Rainbow

DQN and DQN-related algorithms are state-of-the-art on Atari arcade games. However, Flatland poses more complex challenges than the ones faced in Atari, first of all being a multi-agent environment [23]. Our approach is limited in the modifications done to Rainbow to adapt it to a MAS. Therefore, we suggest that applying different algorithms from literature, that are suited for MARL and deal with the non-stationarity of the environment, could lead to substantial improvements. We refer both to algorithms derived from DQN than others, such as Actor-Critic.

Reward

In every RL problem, reward definition is key to represent the goal idea and allow the agent to achieve its goal. Shaping reward inevitably influences the behavior of the learning agent. In a multi-agent setting, defining reward is complex, in a cooperative environment such as Flatland, each agent possesses its own goal, that is reaching the target as fast as possible, and all of them concur in a race to achieve a shared goal, let the maximum number of agents reach their target. It is easy to see how balancing individuality and cooperation is here vital to obtain the best outcome.
In our approach, we decided to assign a positive reward to the agents independently of each other. Being the task maximizing the number of trains that can get to the destination, then giving a single reward seemed to us more appropriate than a global positive one assigned only in case all agents

complete.
Another issue in Flatland is *sparsity of rewards*, an agent receives a positive reward only at the target, that is, after choosing a long sequence of actions, with no other feedback in its way to the target, except a simple penalty on time, that doesn't depend on the action picked.
We propose that adding *local rewards*, for example in case of overcoming conflicting situation, could be beneficial in helping the agent to learn better.

Generalization

A core challenge in RL is *generalization*, an agent that, after a long training, performs very well on a specific environment, it is not sure to perform as well in a slightly different environment. In Flatland the problem consists first in finding a good policy that generalizes on environments of the same size (width x height) and different inner structures, such as number of cities, number of rails between cities and placement of these inside the map. Secondly, a good policy must be able to generalize correctly for different environment sizes and different number of agents. This poses crucial difficulties in terms of scalability of the solution and computational effort.
In our approach we started investigating the tasks on simple grids to take into account this problem only later.

Time

The idea of time is crucial in RL, that deals with agents facing complex decision-making problems. In Flatland, timing is vital to avoid conflicts and it is a key element to be considered during prediction, since agents path duration varies depending on different speed profiles.
The Graph Observations approach was based on the idea of prediction, namely, for how many cells in time we will be aware of the agent position. However, our idea of prediction is *single*, meaning, not dependent on other agents predictions. A possible improvement would be to consider a more global idea of prediction in order to take into consideration the interactions

between agents in time.

Stochasticity

In the perspective of Multi-Agent Reinforcement Learning, the presence of stochastic events that are not determined by the agents, introduces an important challenge [49] [50]. Stochasticity in Flatland is represented by occurrence of malfunctions, determined by a Poisson process parameterized with a certain *malfunction rate*. The occurrence of malfunctions introduce an additional problem to the detection of conflicts. In the standard situation, conflicts rise when two paths overlap, but malfunctions add a stochastic and unpredictable factor that forces agents to re-compute their routes in order to avoid conflicts on rails that contain stuck agents. The first solution that we propose to avoid these situations is that each agent must be aware of multiple paths that lead to the target, to choose an alternative in all the cases a malfunction happens and the predefined path is disrupted because a rail becomes temporarily unavailable.
In the Graph Observations, a limitation in our approach is the consideration of shortest paths only. So we propose to enrich the algorithmic part with a controller that performs a more sophisticated search on the railway graph. In general, the algorithm that computes the alternative paths must be updated to consider not only the structure of map but also the position of other agents to avoid those that are affected by a failure.

More generally, other approaches can be tried and different techniques can be included, such as decomposition of the end-to-end learning problem into submodules to speed up learning as proposed in [47] or the mix of reinforcement learning with supervised or causal learning, as suggested in [48].
In the following paragraphs we detail some more general research directions and possible improvements.

Graph Neural Networks (GNNs)

Graph Neural Networks (GNNs) are models that try to capture existing relationships within graph data which contains rich relation information between elements [46]. These models capture dependence via message passing between the nodes of the graphs and differently from standard neural networks, they retain a state that can represent information from its neighborhood with arbitrary depth.
In literature there are many proposed architectures to deal with graph data using GNNs.

Communication among agents and action negotiation

In multiagent reinforcement learning, some form of communication between agents is often used [44] [45], especially in cooperative settings where a common goal is present and agents need to coordinate themselves to achieve it. Even in Flatland, where the goal is to maximize the number of agents reaching the targets in the minimum amount of time, some form of communication can be implemented. For example, one agent could potentially be aware of the intentions of the other agents (e.g. next planned actions) and act accordingly. In some settings, also adding a *master agent* that controls everybody at once proved to be a successful resolution.
A further mechanism to avoid undesired scenarios is to add another round of action negotiation following the action selection phase. When all the agents have chosen their actions and a complete knowledge is available, illicit situations such as unavoidable conflicts can be detected and a new action selection phase can be performed pushing the agents to pick alternative actions to avoid these invalid cases.

Operation research methods

Vehicle Rescheduling Problem (VRSP) is a combinatorial optimization problem and like others of this kind can be tackled using Operation Research

(OR), Constraint Programming (CP) or a combination of the two [42] [43]. Adapting Flatland tasks to these formalisms could give us a different way to analyze the problem under a mathematical perspective and produce a classical algorithm.

Having available such an algorithm would provide us with a baseline for RL algorithms or other types of learning, such as *imitation learning*, where we learn a policy by observing first the actions picked following the baseline.

Conclusion

In this work we explored Reinforcement Learning principles and its representative elements with a focus on the subdomain of multi-agent systems and its challenges. We explained the classical approaches to solve RL problems and we focused on Deep RL, where methods include neural networks as function approximators to represent some aspects of the RL framework and we detailed one of the most influential algorithms in the Deep RL literature, DQN and its more performing variants.
Next, we described Flatland, a multi-agent railway simulation that built the foundation for the environment on which this study is based, and we applied DQN and Rainbow on a series of diversified tasks derived from the formulation of the Vehicle Rescheduling Problem (VRSP) within this railway environment, where the common goal is to optimize the traffic flow within the network.

To overcome the computational difficulties of solving the VRSP on realistic environments using deterministic algorithms and heuristics, we proposed a series of alternative approaches based on Deep Reinforcement Learning. These approaches were applied on a series of tasks of increasing difficulty to investigate the limitations and the challenges of this learning framework, together with its potentiality for success.
In the *navigation task*, where an agent from a random position on a map must reach its target station, we showed how DRL with DQN can achieve optimality and a comparable performance to a standard algorithm, such as Dijkstra's algorithm on graphs. The challenge faced in this task is to find the

best approximation for the environment state (observation), in order for the agent to reach its goal. In our simple case, the key idea is a reward shaping that reinforces a positive reward only when the target is reached and assign a negative reward in all the other steps, in order to penalize an agent for its time spent in the map, so as to speed up its run to the target.
The crucial point in this observation is to feed the agent data about its *distance* to the target, that changes at every navigation choice.
In this framework, it would be interesting to extend the work to identify ways to speed up learning by finding the best observation representation for the agent.

On the other hand, identifying a successful methodology that could lead to good outcomes in a multi-agent setting showed more difficulties.
In the *conflicts avoidance task* the main challenges are derived from the complexity that is typical of multi-agent systems. In this scenario, learning can become chaotic, since observations for one agent have to take into consideration also other agents, in a vision that is not stationary anymore. Here we implemented a first simplified possible approach that leverages the structure of the railway network as a graph, by decomposing the task in two levels, where only one of them is controlled by DRL algorithms, producing results that are open to many future improvements.
In the second approach we developed instead a particular representation of the rails as resources and their occupation in time through a bitmap, that was also the chosen data structure to express the probability of conflicts between agents. In this case, the positive results opened to more possibilities of experimenting with different parameters and settings.

Appendix A

Graph Observations

This is the main method that is called whenever observation must be computed for one agent.

```
def get(self, handle: int = 0) -> {}:
        """
        Returns obs for one agent, obs are a single array of concatenated
            values representing:
        - occupancy of next prediction_depth cells,
        - forks,
        - targets,
        - priority,
        - number of malfunctioning agents (encountered),
        - number of agents that are ready to depart (encountered).
        :param handle: agent id
        :return: a graph observation for agent handle
        """

        agents = self.env.agents
        agent = agents[handle]

        # Occupancy
        occupancy, conflicting_agents = self._fill_occupancy(handle)
        # Augment occupancy with another one-hot encoded layer: 1 if this
            cell is overlapping and the conflict span was already entered
            by some other agent
```

```
second_layer = np.zeros(self.max_prediction_depth, dtype=int) #
    Same size as occupancy
for ca in conflicting_agents:
    if ca != handle:
        # Find ts when conflict occurred
        ts = [x for x, y in enumerate(self.cells_sequence[handle])
            if y[0] == agents[ca].position[0] and y[1] ==
            agents[ca].position[1]] # Find index/ts for conflict
        # Set to 1 conflict span which was already entered by some
            agent - fill left side and right side of ts
        if len(ts) > 0:
            i = ts[0] # Since the previous returns a list of ts
            while 0 <= i < self.max_prediction_depth:
                second_layer[i] = 1 if occupancy[i] > 0 else 0
                i -= 1
            i = ts[0]
            while i < self.max_prediction_depth:
                second_layer[i] = 1 if occupancy[i] > 0 else 0
                i += 1

occupancy = np.append(occupancy, second_layer)

# Bifurcation points, one-hot encoded layer of predicted cells
    where 1 means that this cell is a fork
# (globally - considering cell transitions not depending on agent
    orientation)
forks = np.zeros(self.max_prediction_depth, dtype=int)
# Target
target = np.zeros(self.max_prediction_depth, dtype=int)
for index in range(self.max_prediction_depth):
    # Fill as 1 if transitions represent a fork cell
    cell = self.cells_sequence[handle][index]
    if cell in self.forks_coords:
        forks[index] = 1
    if cell == agent.target:
        target[index] = 1
```

```
# Speed/priority
is_conflict = True if len(conflicting_agents) > 0 else False
priority = assign_priority(self.env, agent, is_conflict)
max_prio_encountered = 0
if is_conflict:
    conflicting_agents_priorities = [assign_priority(self.env,
        agents[ca], True) for ca in conflicting_agents]
    max_prio_encountered = np.min(conflicting_agents_priorities) #
        Max prio is the one with lowest value

# Malfunctioning obs
# Counting number of agents that are currently malfunctioning
    (globally) - experimental
n_agents_malfunctioning = 0 # in TreeObs they store the length of
    the longest malfunction encountered
for a in agents:
    if a.malfunction_data['malfunction'] != 0:
        n_agents_malfunctioning += 1 # Considering ALL agents

# Agents status (agents ready to depart) - it tells the agent how
    many will appear
n_agents_ready_to_depart = 0
for a in agents:
    if a.status in [RailAgentStatus.READY_TO_DEPART]:
        n_agents_ready_to_depart += 1 # Considering ALL agents

# shape (prediction_depth * 4 + 4, )
agent_obs = np.append(occupancy, forks)
agent_obs = np.append(agent_obs, target)
agent_obs = np.append(agent_obs, (priority, max_prio_encountered,
    n_agents_malfunctioning, n_agents_ready_to_depart))

return agent_obs
```

Here instead we show how priority is computed.

```
def assign_priority(env, env_agent, conflict):
    """
    Assign priority in this way:
    - if agent is READY_TO_DEPART or DONE, return 0
    - if agent is ACTIVE:
        - if no conflict was predicted, return 0 (max prio)
        - if conflict was predicted,
        return priority as function of distance to target and speed
    :param env_agent:
    :param conflict:
    :return:
    """
    if env_agent.status is not RailAgentStatus.ACTIVE:
        return 0

    if not conflict:
        return 0
    else:
        max_distance = distance_on_rail((0,0), (env.height-1, env.width-1))
        min_distance = 0
        min_speed = 0.25
        # Use Euclidean distance
        priority = distance_on_rail(env_agent.position, env_agent.target)
        priority /= env_agent.speed_data['speed']
        # Normalize priority
        priority = np.around(priority / (max_distance / min_speed),
            decimals=3)
        return priority
```

Bibliography

[1] Richard S. Sutton, Andrew G. Barto. 2018.
Reinforcement Learning: An Introduction.
The MIT Press, Second Edition.

[2] Ian Goodfellow, Yoshua Bengio, Aaron Courville. 2016.
Deep Learning. The MIT Press.

[3] D. Silver. *UCL Course on RL*.
http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html.
Last visited: 5 January 2020.

[4] D. Farinelli. 2020. "Apprendimento con rinforzo applicato allo scheduling dei treni per la Flatland challenge". *AMSLaurea, AlmaDL: University of Bologna Digital Library*.

[5] V. François-Lavet, R. Islam, J. Pineau, P. Henderson, M.G. Bellemare. 2018. "An Introduction to Deep Reinforcement Learning". *Foundations and Trends in Machine Learning*: Vol. 11, No 3-4.

[6] Y. Li. 2018. "Deep Reinforcement Learning". *arXiv preprint arXiv:1810.06339*.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller. 2013. "Playing Atari with Deep Reinforcement Learning". *arXiv preprint arXiv:1312.5602*.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A.A. Rusu, J. Veness, M.G. Bellemare, A. Graves, M. Riedmiller, A.K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg. & D. Hassabis. 2015. "Human-level control through deep reinforcement learning". *Nature*: Vol. 518, 529–533(2015).

[9] D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, & D. Hassabis. 2016. "Mastering the game of Go with deep neural networks and tree search". *Nature*: Vol. 529, 484–489(2016).

[10] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel. & D. Hassabis. 2017. "Mastering the game of Go without human knowledge". *Nature*: Vol. 550, 354–359(2017).

[11] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, D. Hassabis. 2018. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". *Science*: Vol. 362, Issue 6419, pp. 1140-1144.

[12] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, D. Silver. 2019. "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model". *arXiv preprint arXiv:1911.08265.*

[13] S. Levine, C. Finn, T. Darrell, P. Abbeel. 2016. "End-to-end training of deep visuomotor policies". *Journal of Machine Learning Research.* 17(39): 1-40.

[14] D. Ghandi, L. Pinto, A. Gupta. 2017. "Learning to Fly by Crashing". *arXiv preprint arXiv:1704.05588*

[15] Y. You, X. Pan, Z. Wang, C. Lu. 2017. "Virtual to Real Reinforcement Learning for Autonomous Driving". *arXiv preprint arXiv:1704.03952.*

[16] R. Paulus, C. Xiong, R. Socher. 2017. "A Deep Reinforced Model for Abstractive Summarization". *arXiv preprint arXiv:1705.04304.*

[17] J. Li, W. Monroe, A. Ritter, M. Gallery, J. Gao, D. Jurafsky. 2016. "Deep Reinforcement Learning for Dialogue Generation". *arXiv preprint arXiv:1606.01541.*

[18] Y. Rao, J. Lu, J. Zhou. 2017. "Attention-Aware Deep Reinforcement Learning for Video Face Recognition". *2017 IEEE International Conference on Computer Vision (ICCV).*

[19] J. Li, P. B. Mirchandani, D. Borenstein. 2007. "The vehicle rescheduling problem: Model and algorithms". *Networks.* 50 (3): 211–229

[20] R. Bellman. 1957. "A Markovian decision process". *Journal of Mathematics and Mechanics*: 679-684.

[21] C.J.C.H. Watkins. 1989. "Learning from Delayed Rewards". *Ph.D. thesis, Cambridge University.*

[22] C.J.C.H. Watkins, P. Dayan. 1992. "Q-Learning". *Machine Learning*: Vol. 8, 3-4, pp. 279-292.

[23] T.T. Nguyen, N.D. Nguyen, S. Nahavandi. 2019. "Deep Reinforcement Learning for Multi-Agent Systems: A Review of Challenges, Solutions and Applications". *arXiv preprint arXiv:1812.11794.*

[24] L. Busoniu, B. De Schutter, R. Babuska. 2010. "Multi-agent Reinforcement Learning: An Overview". *Innovations in Multi-Agent Systems and Applications*: Vol 310, pp.183-221.

[25] M.L. Littman. 1994. "Markov games as a framework for multi-agent reinforcement learning". *Proceedings of the eleventh international conference on machine learning*: Vol. 157, pp. 157-163.

[26] OpenAI. OpenAI Spinning Up.
https://spinningup.openai.com/en/latest/index.html.
Last visited: 14 January 2020.

[27] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, D. Silver. 2017. "Rainbow: Combining Improvements in Deep Reinforcement Learning". *arXiv preprint arXiv:1710.02298*.

[28] H. van Hasselt. 2010. "Double Q-learning". *Advances in Neural Information Processing Systems*: Vol 23, pp. 2613-2621.

[29] H. van Hasselt, A. Guez, D. Silver. "Deep Reinforcement Learning with Double Q-Learning"- *arXiv preprint arXiv:1509.06461*.

[30] T. Schaul, J. Quan, I. Antonoglou, D. Silver. 2015. "Prioritized experience replay". *Proceedings of ICLR*.

[31] Z. Wang, T Schaul, Hessel M., H. van Hasselt, M. Lanctot, N. de Freitas. 2016. "Dueling network architectures for deep reinforcement learning". *Proceedings of The 33rd International Conference on Machine Learning, 1995-2003*.

[32] Andrew Melnik. *Deep Q-Network & Dueling network architectures for deep reinforcement learning*. https://youtu.be/XjsY8-P4WHM.
Last visited: 19 January 2020.

[33] J.F. Hernandez-Garcia, R.S. Sutton. 2019. "Understanding Multi-Step Deep Reinforcement Learning: A Systematic Study of the DQN Target". *arXiv preprint arXiv:1901.07510*.

[34] M.G. Bellemare, W. Dabney, R. Munos. 2017. "A distributional perspective on reinforcement learning". *ICML*.

[35] M. Fortunato, M.G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, S. Legg. 2017. "Noisy networks for exploration". *CoRR*: abs/1706.10295.

[36] AIcrowd. *Flatland Challenge*.
https://www.aicrowd.com/challenges/flatland-challenge.
Last visited: 7 January 2020.

[37] AIcrowd. *Crowdsourcing AI to solve real-world problems*. https://www.aicrowd.com/ Last visited: 7 January 2020.

[38] AIcrowd, Flatland community. *Flatland 2.1.10 Documentation*. http://flatland-rl-docs.s3-website.eu-central-1.amazonaws.com. Last visited: 8 January 2020.

[39] E.W. Dijkstra. 1959. "A Note on Two Problems in Connexion with Graphs". *Numerische Mathematik*: Vol 1, 269–271.

[40] Kai Arulkumaran (Kaixhin). *Kaixhin/Rainbow*. https://github.com/Kaixhin/Rainbow Last visited: 12 January 2020.

[41] Alexander Lenail. *NN-SVG Publication-ready NN-architecture schematics*. http://alexlenail.me/NN-SVG/index.html Last visited: 14 January 2020.

[42] J.N. Hooker, W.J. van Hoeve. 2017. "Constraint programming and operations research". *Constraints*: Volume 23, Issue 2, pp 172–195.

[43] J.N. Hooker, F. Rossi, P. van Beek, T. Walsh. 2006. "Operations Research Methods in Constraint Programming". *Handbook of Constraint Programming*. Elsevier.

[44] M.S. Zaiem, E. Bennequin. 2019. "Learning to Communicate in Multi-Agent Reinforcement Learning: A Review". *arXiv preprint arXiv:1911.05438.*

[45] J.N. Foerster, Y.M. Assael, N. de Freitas, S. Whiteson. 2016. "Learning to Communicate with Deep Multi-Agent Reinforcement Learning". *arXiv preprint arXiv:1605.06676.*

[46] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, M. Sun. 2019. "Graph Neural Networks: A Review of Methods and Applications". *arXiv preprint arXiv:1812.08434.*

[47] A. Melnik, S. Fleer, M. Schilling, H. Ritter. 2019. "Modularization of End-to-End Learning: Case Study in Arcade Games". *arXiv preprint arXiv:1901.09895.*

[48] A. Melnik, L. Bramlage, H. Voss, F. Rossetto, H. Ritter. 2019. "Combining Causal Modelling and Deep Reinforcement Learning for Autonomous Agents in Minecraft". *4th Workshop on Semantic Policy and Action Representations for Autonomous Robots at IROS 2019, Macau.*

[49] N.L. Kuang, C.H.C. Leung, V.W.K. Sung. 2019. "Stochastic Reinforcement Learning". *arXiv preprint arXiv:1902.04178.*

[50] R. Ceren. 2019. "Optimal Decision-Making in Mixed-Agent Partially Observable Stochastic Environments via Reinforcement Learning". *arXiv preprint arXiv:1901.01325.*