ALMA MATER STUDIORUM - UNIVERSITY OF BOLOGNA

Deep Learning Project Report

Flatland Challenge: a Deep Reinforcement Learning approach

Professor:

Andrea Asperti

Students:

Diego Chinellato - diego.chinellato@studio.unibo.it

Giorgia Campardo - giorgia.campardo@studio.unibo.it

Academic Year

2020/2021

Abstract

The Flatland Challenge is a competition which aims at promoting progress in a multi-agent setting for the vehicle re-scheduling problem (VRSP). The challenge addresses a real-world problem faced by many transportation and logistics companies around the world (such as the Swiss Federal Railways, SBB), that is, the automatic management of a large scale railway network. Flatland offers a simplified 2D railway simulation where different tasks related to VRSP must be solved.

This report describes a Deep Reinforcement Learning (DRL) approach to the Flatland Challenge: several DRL algorithms and techniques are considered and have been implemented to addreess this problem, along with an original piece of work specifically tailored for the VRSP problem.

Contents

Introduction

At the core of this challenge lies the general vehicle re-scheduling problem (VRSP) proposed by Li, Mirchandani and Borenstein in [1]:

> *The vehicle rescheduling problem (VRSP) arises when a previously assigned 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. The VRSP can be approached as a dynamic version of the classical vehicle scheduling problem (VSP) where assignments are generated dynamically.*

The simulation of the above problem offered by Flatland consists in a 2D grid world, where each cell defines where an agent (i.e. train) can be located and how, if possible, it can move to a neighbouring cell. Different agents may have different objectives (i.e. target train stations) and in the general setting they must collaborate to reach their respective objectives as soon as possible. The general goal is to have all the agents reaching their target station as soon as possible; in order word, the agent that has to be developed must be capable of minimizing the arrival time of each single agent so to minimize the total waiting time.

All the code developed for this project is available in two separate GitHub repositories: the PyAgents repository contains problem-agnostic implementations of DRL algorithms discussed in this report, whereas the Flatland-RL repository implements an agent (using the first repository) for solving the Flatland Challenge.

Chapter 1

Flatland

1.1 Environment

In this first chapter we introduce the environment in which the project will be carried out. The *Flatland Environment* [2] is a 2D rectangular grid environment of arbitrary width and height, where the most primitive unit is a cell. Each cell has the capacity to hold a single agent (train). An agent in a cell can have a discrete orientation direction which represents the cardinal direction the agent is pointing to. An agent can move to a subset of adjacent cells. The subset of adjacent cells that an agent is allowed to transition to is defined by a 4-bit transition map representing possible transitions in 4 different directions.

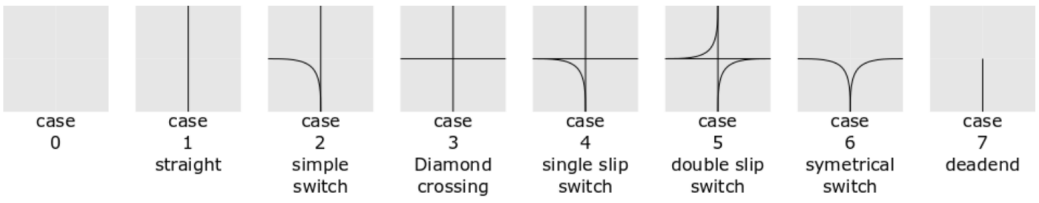

Figure 1.1: 8 unique cells that can represent any real world railway network in the flatland environment.

Agents can only travel in the direction they are currently facing. Hence, the permitted transitions for any given agent depend both on its position and on its direction. Moreover malfunctions are implemented to simulate delays by

stopping agents at random times for random duration. Train that malfunction can't move for a random, but known, number of steps.

1.2 Actions

The trains in Flatland have strongly limited movements, in order to simulate a real railway situation as closely as possible. The action space is discrete(5):

- DO_NOTHING: if the agent is already moving, it continues moving. If it is stopped, it stays stopped. Special case: if the agent is at a dead-end, this action will result in the train turning around.
- MOVE_LEFT: this action is only valid at cells where the agent can change direction towards the left. If chosen, the left transition and a rotation of the agent orientation to the left is executed. If the agent is stopped, this action will cause it to start moving in any cell where forward or left is allowed.
- MOVE_FORWARD: the agent will move forward. This action will start the agent when stopped. At switches, this will chose the forward direction.
- MOVE_RIGHT: the same as deviate left but for right turns.
- STOP_MOVING: stops the agent from moving.

Flatland is a discrete time simulation, i.e. it performs all actions with constant time step. A single simulation step synchronously moves the time forward by a constant increment, thus enacting exactly one action per agent per time step.

1.3 Observations

At each time step the Flatland environment has to provide the current state of the environment to the agent. In the library, three observations that represent the environment's state are provided.

1.3.1 Global observation

The global observation is the simplest one: every agent is provided a global view of the full Flatland environment. This can be compared to the full, raw-pixel data used in Atari games. The size of the observation space is h × w × c, where h is the height of the environment, w is the width of the environment and c is the number of channels of the environment, 23 in total. The first 16 channels are the transition maps, which provide a unique value for each type of transition map and its orientation using a 16bits encoding where the bits have the following meaning: NN NE NS NW EN EE ES EW SN SE SS SW WN WE WS WW. For example, the binary code 1000 0000 0010 0000, represents a straight where an agent facing north can transition north and an agent facing south can transition south and no other transitions are possible. Then there are 5 channels reserved for the agent states:

- **Channel 0**: a one-hot representation of the self agent position and direction
- **Channel 1**: other agents' positions and direction
- **Channel 2**: self and other agents' malfunctions
- **Channel 3**: self and other agents' fractional speeds
- **Channel 4**: the number of other agents ready to depart from that position

The last 2 channels represent the agent targets, containing respectively the position of the current agent target, and the positions of the other agents' targets. The positions of the targets of the other agents is a simple 0/1 flag,

therefore this observation doesn't indicate where each other agent is heading to.

1.3.2 Local grid observation

The local grid observation is very similar to the global observation, where we only replace h and w by agent specific dimensions. The agent is always situated at the position (0, (w + 1)/2) within the observation grid and the observation grid is rotated according to the agent's direction such that the full height h of the observation grid is in front of the agent. The initial local grid view provides the same channels as the initial global view introduced above. This observation space offers benefits over the global view, mostly by reducing the amount of irrelevant information in the observation space. Global navigation with this local observation would be impossible if no general information about the target location were given (especially when the target is outside of view). Therefore a distance map for every agent-target is computed and provided as an additional channel, which allows for a sense of direction without the need of a global view.

1.3.3 Tree observation

The tree observation exploits the fact that a railway network is a graph and thus the observation is only built along allowed transitions in the graph. The observation is generated by spanning a 4 branched tree from the current position of the agent. Each branch follows the allowed transitions (backward branch only allowed at dead-ends) until a cell with multiple allowed transitions is reached. Here the information gathered along the branch is stored as a node in the tree. It is important to note that the tree observation is always build according to the orientation of the agent at a given node. This means that each node always has 4 branches coming from it in the directions Left, Forward, Right and Backward. The tree always has the same number of nodes for a given tree depth, this is achieved by filling nodes where there are no possibilities with

-inf. Each node is filled with information gathered along the path to the node, in particular it contains 12 features:

- **Channel 0**: if own target lies on the explored branch the current distance from the agent in number of cells is stored.
- **Channel 1**: if another agents target is detected the distance in number of cells from current agent position is stored.
- **Channel 2**: if another agent is detected the distance in number of cells from current agent position is stored.
- **Channel 3**: possible conflict detected - this relies on a predictor which anticipates where agents will be in the future. Possible values are the distance in number of cells from current agent position when another agent predicts to pass along this cell at the same time as the agent, 0 otherwise.
- **Channel 4**: if a not usable switch (for the agent) is detected, its distance is stored. An unusable switch is a switch where the agent does not have any choice of path (i.e. the agent is blocked), but other agents coming from different directions might.
- **Channel 5**: This feature stores the distance (in number of cells) to the next node (e.g. switch or target or dead-end).
- **Channel 6**: minimum remaining travel distance from node to the agent's target given the direction of the agent if this path is chosen.
- **Channel 7**: number of agents going in the same direction found on path to node.
- **Channel 8**: number of agents going in the opposite direction found on path to node.
- **Channel 9**: malfunctioning/blocking agents, returns the number of time steps the observed agent will remain blocked.

- **Channel 10**: slowest observed speed of an agent in the same direction, 1 if no agent is observed.
- **Channel 11**: number of agents ready to depart but no yet active.

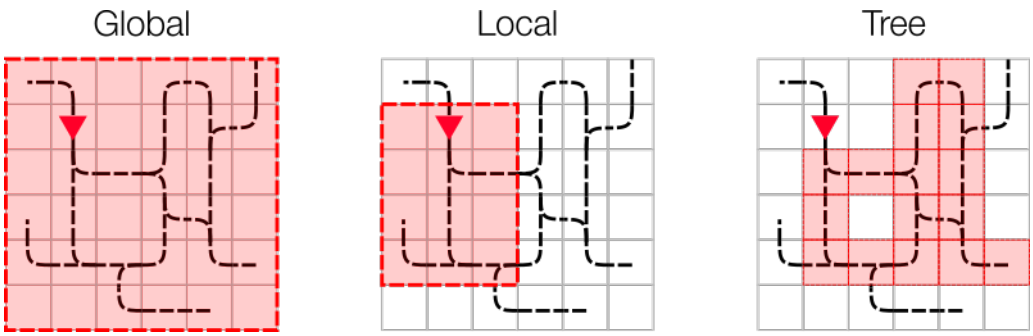

Figure 1.2: A visual summary of the three provided observations

1.4 Rewards

Each agent receives a combined reward consisting of a local and a global reward signal. For every time step t, an agent i receives a local reward $r_l^i = -1$ if it is moving or stopped along the way and $r_l^i = 0$ if it has reached its target location. In addition, a configurable penalty r_p^i is received if the move is illegal, i.e. the action cannot be executed, by default it's set to 0. If all agents have reached their targets at time step t, a global reward $r_g = 1$ is awarded to every agent; otherwise all agents get $r_g^t = 0$. Overall, every agent i receives the reward

$$r^i(t) = \alpha r_l^i(t) + \beta r_g(t) + r_p^i(t) \in [-\alpha - 2, \beta]$$

where α and β are parameters for tuning collaborative behavior. Therefore, the reward creates an objective of finishing the episode as quickly as possible in a collaborative way. At the end of each episode, the return of each agent i is given by the sum over the time step rewards:

$$g_i = \sum_{t=1}^{T} r_i(t)$$

where T is the time step when the episode terminated. [3]

Chapter 2

Theoretical preliminaries

This chapter deals with the mathematical background that is involved in the work that has been carried out for this project.

2.1 Reinforcement Learning setting

Reinforcement Learning (RL) problems involve learning what to do - that is, how to map situations into actions - so as to maximize a numerical reward signal [4]. In a RL problem there are two main entities at work: the *learner*, which is called the *agent*, and the *environment*, which is everything else that the agent interacts with. More specifically, the agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, ...$ At each time step t, the agent receives some representation of the environment's state, $S_t \in S$, where S is the set of possible states, and on that basis selects an action, $A_t \in \text{ACTIONS}(S_t)$, where $\text{ACTIONS}(S_t)$ is the set of actions available in state S_t. One time step later, partly also as a consequence of its action, the agent receives a numerical reward, $R_{t+1} \in \mathbb{R}$, and finds itself in a new state, S_{t+1}. Figure 2.1 diagrams the agent–environment interaction.

At each time step, the agent implements a mapping from states to probabilities of selecting each possible action. This mapping is called the agent's policy and is denoted π_t, where $\pi_t(a|s)$ is the probability that $A_t = a$ if $S_t = s$. [4]. Reinforcement learning algorithms describe how the agent changes its

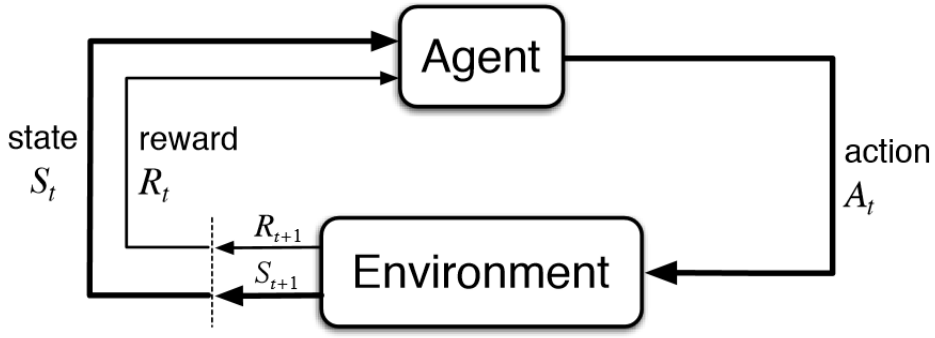

Figure 2.1: The RL setting.

policy as a consequence of learning. The ultimate goal for the agent, then, is to learn the optimal policy π^* which maximizes the expected (discounted) cumulative return G_t, defined as

$$\pi^* = \arg\max_{\pi} G_t$$

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 T_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where $\gamma \in [0, 1)$ is called the discount rate and is an hyperparameter controlling the relative importance given to rewards closer in time (higher when $\gamma \to 0$) w.r.t. rewards that are further in time (higher when $\gamma \to 1$).

The formal model used to represent a RL problem is called a Markov Decision Process (MDP). A (finite) MDP is a tuple (S, A, f, ρ) where S is the finite set of environment states, A is the finite set of agent actions, $f : S \times A \times S \to [0, 1]$ is the state transition probability function, and $\rho : S \times A \times S \to R$ is the reward function. [4]. Indeed, MDPs assume there is only one agent interacting with the environment. For our purposes, this indeed does not suffice as the Flatland Challenge involves the concurrent management of many trains (agents); in other words, our problem is a Multi Agent Reinforcement Learning (MARL) problem. The generalization of MDPs to the multi agent case is called stochastic game, and it is pretty straightforward. A stochastic game is a tuple $(S, A_1, ..., A_n, f, \rho_1, ..., \rho_n)$ where n is the number of agents, S is the finite set of (joint) environment states, $A_i, i = 1, ..., n$ are the finite sets of actions available

to the agents, yielding the joint action set $\mathbf{A} = A_1 \times \times A_n$, $f : S \times \mathbf{A} \times S \to [0, 1]$ is the state transition probability function, and $\rho_i : S \times \mathbf{A} \times S \to R, i = 1, ..., n$ are the reward functions of the agents [5]. In other words, states and actions becomes joins states and joint actions of all the agents, with different reward functions for each agent and the transition function only need to take into account the new type of states and actions. Without loss of generality, in the following we will mostly refer to the single agent case (as it can be easily extended to the multi agent case).

2.1.1 Value functions

A central concept in RL is the idea of value functions, i.e. functions of states that measure in some way how good it is for the agent to find itself in a certain state. Indeed, this "goodness" of the state is measured in terms of expected future returns, which in turn depends on the policy of the agent (since the policy essentially dictates how the agent chooses its actions).

There are two main value functions that we consider. The first is called the state-value function, denoted $V_\pi(s)$, and is defined as the expected return for an agent that starts in state s and follows policy π from that state on. Formally:

$$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

Similarly, we define the action-value function, denoted $Q_\pi(s, a)$, as the expected return for an agent that starts in state s, takes action s and follows policy π thereafter. Formally:

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

Indeed, there is a relationship between the two value functions:

$$V_\pi(s) = \sum_{a \in A} \pi(s, a) \cdot Q_\pi(s, a)$$

Many RL algorithms involve estimating, either directly or indirectly, one of the two value functions. Note that each value function implicitly defines a policy, and this idea will be central in Q-Learning.

2.1.2 Flavours of RL algorithms

Although there many different types of RL algorithms, most of them can be divided in one of two categories:

1. Model-based methods: this family of methods works by building through learning an internal model of the environment, i.e. a "black box" that can be used to make predictions about the next state and next reward given a state and an action, without the agent actually having to perform the action.

2. Model-free methods: as the name suggests, this family of algorithms doesn't build an internal representation of the external environment, rather they only sample the environment at every time step and use the obtained samples to build (learn) a policy. This policy can be obtained usually in two different ways: with a value-based approach, the agent doesn't learn the policy directly, rather it estimates a value function and then once the value function is available the policy is implicit, i.e. it suffices to choose the action with the highest value (according to the estimated value function). Conversely, with a policy-based approach the agent avoids estimating the value function and instead it tries to learn the optimal policy directly.

Moreover, in RL we also identify two types of problem: first, there is the policy evaluation or prediction problem, which is the problem of estimating the value function V_π for a given policy π; another thing is the control problem, which involves finding the optimal policy π^*. Without going into the details, it suffice to say that most algorithms focus on solving the prediction problem and then use some common technique (usually Generalized Policy Iteration) to also solve the control problem. For this project, we will mainly focus on model-free, value-based methods for the control problem.

2.2 Q-learning

Q-learning was one of the earliest breakthroughs in RL. First described by Watkins in [6], it is a model-free, value-based, off-policy TD control algorithm that learns a policy $\pi \approx \pi^*$ by leveraging Bellman's optimality equations, defined as as:

$$V^*(s) = \max_\pi V_\pi(s)$$

$$Q^*(s,a) = \max_\pi Q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma V^*(S_{t+1}) | S_t = s, A_t = a]$$

The core idea of Q-learning is to directly approximate the optimal action-value function Q^*. It does so by maintaining a table Q for all $(state, action)$ pairs, and then at each time step t it chooses an action A_t using a policy derived from Q (e.g. choosing the action with highest action value), it executes A_t observing a reward R_{t+1} and next state S_{t+1} and finally updates $Q(S_t, A_t)$ using Bellman's optimality equation (combined with a Temporal Difference error):

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

It has been shown that, under some minimal assumptions (i.e. all state-action pairs have to be updated regularly; it's minimal because any method guaranteed to find the optimal policy must require it), with Q-learning the function Q converges to the optimal Q^* with probability 1.

2.3 DQN

Most of the times the state space is so huge that it's not feasible to represent the Q function by means of a table, like Q-learning does. To solve this issue, we can replace the table with a neural network acting as a universal function approximator; this is the core idea beneath Deep Q Networks (DQN) [7].

Before DQN, Reinforcement learning was known to be unstable or even to diverge when a nonlinear function approximator such as a neural network

is used to represent the Q function, due to e.g. correlations present in the observation (data is not i.i.d.) or the fact that the policy changes (and the data distribution as well). The authors of DQN address this problem by introducing two key improvements. The first is a biologically inspired mechanism called Experience Replay. Consider an approximate value function $Q(s, a; \theta_i)$ that is parametrized using a deep neural network, in which θ_i are the weights. To perform experience replay, the agent's "memories" $e_t = (S_t, A_t, R_{t+1}, S_{t+1})$ are stored in a dataset $\mathcal{D} = \{e_1, ..., e_m\}$. During learning, to apply Q-learning updates a sample (minibatch) of experiences $(s, a, r, s') \sim U(\mathcal{D})$ is drawn at random from the dataset of memory; this greatly helps in reducing correlation between the observation and smoothing the data distribution [7]. Then the Q-learning update at iteration i adopts the following loss function:

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

From this loss, we can also notice the second big improvement of DQN, namely the fact that there are two sets of weights (i.e. two neural networks), θ_i and θ_i^-; the two networks are called respectively online network and target network (because it's used to estimate the TD target). The target network's weights are frozen and only updated every C steps, and are held fixed between individual updates. [7] This update can be done in one of two ways:

- Hard update: simply copy all the weights from the online network to the target network, i.e. set $\theta_i^- \leftarrow \theta_i$.
- Soft update (Polyak average): can be seen as a sort of "weighted" copy since the update is partial, i.e. set $\theta_i^- \leftarrow (1-\tau)\theta_i^- + \tau\theta_i$, where $\tau \in (0, 1]$ is an hyperparameter controlling the weight transfer amount.

This technique of using two different networks leads to a more stable training because the target function remains fixed (at least for a while).

2.3.1 Double DQN

One important issue affecting standard Q-learning (and indeed DQN as well) is the fact that it tends to overestimate action values under certain conditions, mostly due to the use of the max operator in the Bellman optimality equation that is used to perform the Q-learning updates; this leads to unstable training and low quality policies.

A solution to this problem has been proposed both in the context of Q-learning [8] as well as in the context of neural networks [9]. The idea is to reduce this overestimation by decomposing the max operation into action selection and action evaluation, each performed by a different network (i.e., online and target networks). In practice, the difference concerns the computation of the TD error (i.e. the loss function):

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(\mathcal{D})}[r + \gamma Q(s', \arg\max_a Q(S_{t+1}, a; \theta_i); \theta_i^-) - Q(s, a; \theta_i)]$$

Intuitively, in double DQNs we are mitigating the overestimation bias by adding a second estimator, so that chances of both estimators being overoptimistic at the same action are simply lower; in other words, one estimator estimates the best action (i.e. performs action selection), whereas the other estimator evaluates the action values (i.e. performs action evaluation) for the estimated best action, in order to cross out the overestimation bias [9].

2.3.2 Dueling DQN

Another interesting and useful improvement over standard DQN comes in the form of so-called dueling architecture (or Dueling DQN). The core idea of this network architecture is to separate the representation of the scalar state values from the (state-dependent) action advantages [10]. A dueling DQN, as it can be seen in Figure 2.2, differs from a standard DQN only in the very last layer: after a common preprocessor, the network splits in two streams, one for representing the state-value function and the other for the advantage function. The two streams are then combined by means of a special aggregating layer to produce an estimate of the Q-values. Intuitively, with this architecture the

network can better learn which state are (or aren't) valuable, without the need of learning the effect of each action in each state. Moreover, this architecture automatically produces separate estimates for the state-value function and advantage function, without any extra supervision; in other words, no kind of algorithmic modification is required to train a dueling DQN (w.r.t. vanilla DQN).

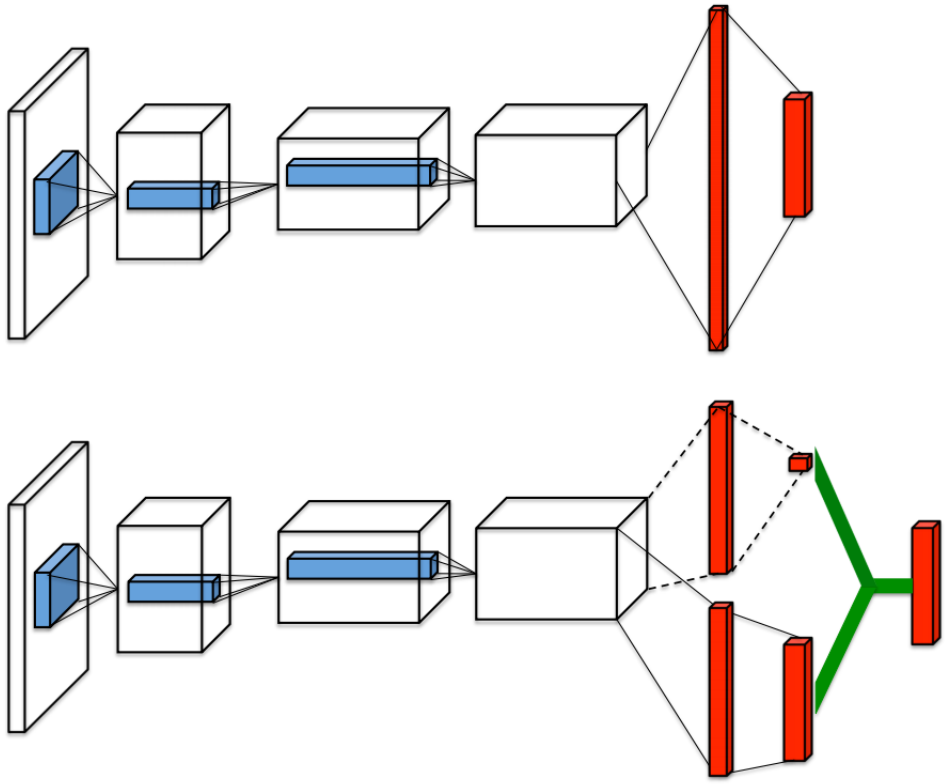

Figure 2.2: Top: a classic, "vanilla" DQN. Bottom: a Dueling DQN, with two separate streams for separately estimating the scalar state-value and the advantages for each action. Image taken from [10].

Formally, the $Q_\pi(s, a)$ function can be decomposed in the sum of two functions $V_\pi(s)$ and $A_\pi(s, a)$ such that $Q_\pi(s, a) = V_\pi(s) + A_\pi(s, a)$. Moreover, given that $V_\pi(s) = \mathbb{E}[Q_\pi(s, a)]$, it follows that $\mathbb{E}[A_\pi(s, a)] = 0$. However, from a practical point of view, the aggregation layer can't simply the sum of the presumed state and advantage values, since they actually aren't reliable estimates of the aforementioned functions and they suffer from lack of identifiability (i.e.,

given a Q value we are not able to recover the corresponding V and A values uniquely), as shown in [10]. To address this issue of identifiability, one idea is to force the advantage function estimator to have zero advantage at the chosen action by using the following mapping in the last layer of the network:

$$Q(s,a) = V(s) + (A(s,a) - \max_{a' \in \mathcal{A}} A(s,a'))$$

An alternative is to use the average rather than the max operator:

$$Q(s,a) = V(s) + (A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s,a'))$$

By using the mean operator rather than the max, the original semantics of $V(s)$ and $A(s,a)$ are lost (they are now off-target by a constant), but on the other hand it increases the stability of the optimization.

2.3.3 Prioritized Experience Replay

As it was highlighted earlier, one of the most important aspect of DQN is the use of Experience Replay (ER), which allows an agent to remember and reuse experiences from the past and in general it can reduce the amount of experience required by the agent to be able to learn. However, given that ER samples pieces of experience uniformly from the memories buffer, they are essentialy replayed at the same frequency that they were originally experienced, no matter their relative significance. An improvement over this basic approach is called Prioritized Experience Replay (PER) [11]. The key idea is that an RL agent can learn more effectively from some transitions than from others. Transitions may be more or less surprising, redundant, or task-relevant; some transitions may not be immediately useful to the agent, but might become so when the agent competence increases [11, 12]. In particular, the authors propose to more frequently replay transitions with high expected learning progress, as measured by the magnitude of their TD error. This prioritization can lead to a loss of diversity, which is dealt with using stochastic prioritization (i.e., keep on sampling experiences probabilistically according to the TD error), and introduce bias, which is corrected using importance sampling.

Chapter 3

Original work

This chapter describes the original work, the modifications and the implementations that have been carried out to tackle the Flatland Challenge.

3.1 Action masking

One of the first issues we faced while tackling the Flatland Challenge was related to legal and illegal moves. In particular, in the Flatland environment each train can execute an action from a subset of all the possible actions, depending on the type of cell the train is in as well as orientation. To exemplify, the action set available to a train in a corridor (i.e. a straight rail without intersections or switches) actually contains a single legal move (moving forward) to choose from, whereas a train at a switch can choose to go in different directions (i.e. several legal moves). Moreover, it's not difficult to compute which move are legal and which aren't, since Flatland is a fully-observable environment; the more difficult part is injecting this type of knowledge into the RL method, so that the agent can learn to avoid illegal moves.

A possible solution would be to add a penalty term to the reward (which is actually already implemented, in the standard Flatland environment), but this is a fragile workaround as it does not guarantee that the agent will learn to avoid illegal moves. A better solution, the one we implemented, is to use a technique called action masking: at each time step and for each agent, the

mask of legal actions is computed; then, this mask is used when the agent is performing action selection to restrict the set of possible moves to the set of the legal moves.

3.2 Shaping the reward signal

Reward shaping attempts to modify the reward signal, provided by the environment to the learning agent, by incorporating additional localized rewards that encourage a behavior consistent with some prior domain knowledge. As long as the intended behavior corresponds to good performance, the learning process will lead to making good decisions. Moreover, because the shaping rewards offer localized advice, the time to exhibit the intended behavior can be greatly reduced. [13, 14]. In other words, rewards shaping should result both in shorter training times as well as in better quality of the resulting learned policies.

The reward signal provided by the Flatland environment has been described in Section 1.4, and can be described by the following function:

$$r^i(t) = \alpha r^i_l(t) + \beta r_g(t) + r^i_p(t)$$

where $r^i_l(t)$ is the local reward, $r_g(t)$ is the global reward, $r^i_p(t)$ is the (configurable) penalty for illegal actions and $\alpha, \beta \in [0, 1]$. Indeed, this signal can be enhanced in several way.

First, given that we are applying action masking to prevent the agent from selecting illegal moves, the $r^i_p(t)$ term of the above equation becomes useless (as it will be always 0 - the agent will never select an illegal action), hence it can be dropped.

Secondly, we introduced additional terms (both penalties as well as rewards) to help the agent learn useful behaviours. These terms are the following:

- Deadlock penalty $p^i_{dead}(t)$. Deadlocks are a serious issue in our setting: a deadlock happens whenever two (or more) trains "bump" on each other, resulting in a situation where both trains can no longer move as they are

blocking each other's way and preventing the possibility of successfully completing an episode (i.e., having all trains arriving at their target station). To discourage the agent from selecting actions that may result in a deadlock, a penalty has been added to the reward signal whenever a deadlock is detected.

- Standing penalty $p^i_{stand}(t)$. The goal of the agent is to make all trains arrive as soon as possible to their relative target station; in other words, a train should be left standing idle (i.e. not moving) at a certain position only when it's really necessary, e.g. to avoid collision with other trains. Hence, we introduced a penalty whenever an idle train is detected. Note that this penalty should be smaller than the deadlock penalty - otherwise we might have an agent that learns that it's better to have trains in a deadlock rather than standing idle.

- "Getting closer" reward $r^i_{closer}(t)$. This reward can be seen as a sort of "discount" on the local reward $r^i_l(t)$. The idea is to given a higher local reward (i.e., closer to 0 than to -1) whenever the agent moves any train closer to its relative target station. This reward is parametrized by a parameter $\theta \in [0, 1]$, and consists in a $+1$ whenever the agent gets closer to its target distance and 0 otherwise.

- Stopping/starting penalties $p^i_{stop}(t), p^i_{start}(t)$. In the real word, starting and stopping a train is a costlier action than a no-op (due to e.g. inertia). Moreover, during our tests we noticed that sometimes the agent would learn to constantly starting and then stopping trains. To discourage this behaviour, we introduced two different penalties whenever a train starts or stops moving. Also, to keep thing loosely related to the real world, the penalty for starting is always slightly higher than the penalty for stopping as indeed more energy is used to start a train rather than to stop it.

3.3 Multi agent iterative training

This approach involves a slight variation of the training procedure. In particular, the idea came from the consideration that it may be more difficult for an agent to learn from scratch how to manage a large rail network with many trains; ideally, if it had some background knowledge coming from a simpler version of the same problem it should be able to learn faster and better. In other words: if the agent is not capable of properly managing n trains, then it's very unlikely that it will be able to manage $n + 1$ trains.

In order to implement this idea, before the actual training with n trains $n - 1$ environments were created with the only difference being the number of trains, ranging from 1 to $n - 1$ trains; the agent is then progressively trained in these environments, starting from the single-agent case up to the $(n-1)$-agents case. In this way, the agent hopefully learns how to manage at first only one train and subsequently more than one until it's ready to start learning the task for n trains, having some background domain knowledge. There is a crucial parameter in this procedure, that is the number of episodes the agent should train on each environment and needs to be tuned during the experiments in order to find the best value maximising the results, taking into account the extra time needed to execute the whole procedure.

During our experiments, it showed to be an effective strategy, leading to better results both in training and in testing, confirming the hypothesis done at the beginning about giving the agent a prior knowledge on simpler environments before starting the real training with n trains and making it able to learn faster and better its task.

3.4 Customizing the observation

Another variation that we implemented in order to (try to) enhance the agent's performance was a modification of the observation. Observations define how our agent will perceive the environment it is in, hence it's is pretty clear that this is a crucial component of any RL-based solution to a problem. The

efficiency and expressiveness of the environment state's encoding may as well separate a good solution from a bad one. Depending on the problem sometimes observations don't leave much space for creativity: for example, in the famous CartPole problem, the observation contains the cart velocity, its angle, and the same information for the pole. Some other problem, instead, involve much more complex observations. For example, when learning to play ATARI games [7], the input observation consists of the pixels values of the game screen. Using suitable image pre-processing and convolutional networks an agent has been proven to be enough to successfully trained and is able to perform well on certain games.

Initially, the network's input was the vectorized encoding of the standard *TreeObservation* provided by Flatland with a tree depth of 2. The encoding is made of 11 features for each node in the tree concatenated together; when the node is *-inf*, meaning there is no possible move in the direction represented by the node, the vector representing this situation is an all-zeros vector; through this procedure the vector obtained with the tree depth of 2 was a vector of 231 features representing the train's state.

With the aim of implementing a more comprehensive and informative observation, we decided to add to the single train's encoding the other trains' encoding but with tree depth of $TD - 1$, leading to a bigger encoding of 341 features as, almost all the times, we used a feasible tree depth of 2. The numbers just stated are the result of this basic formula that is in the program in order to calculate the state shape given a parametric tree depth and number of trains:

$$11 * (\sum_{i=0}^{TD}(4^i) \ + (n-1) * \sum_{i=0}^{TD-1}(4^i))$$

where TD is the tree depth and n is the number of trains in the environment; the summation of 4^i calculates the number of nodes in the tree given the fact each node has four child, one for each direction.

Chapter 4

Experiments and results

This chapter describes how we experimented, evaluated and compared the different agents that have been implemented.

4.1 Experimental setup

In the PyAgents repository it is possible to find the problem-agnostic implementations of DRL algorithms discussed in the previous chapter. Inspired by the work of Tensorflow Agents, we reimplemented the necessary classes, using Tensorflow [15] for the training loop and the loss computation.

In order to carry out the experiments, we used a computer in our possession with a discretely powerful GPU; however, despite our setup we faced several difficulties in carrying on experiments, especially in complex environments with more than three trains in the map where the simulation becomes computationally expensive. This didn't come as a surprise: in DRL (and in general in all applications of DL), training involves dozens of GPUs and millions of frames during entire days of continuous learning [16, 17], thus resulting almost prohibitive for small groups.

Weights & Biases

In order to evaluate different combinations of hyperparameters and strategies we need a tool to store, track, share and effectively compare different runs

Parameter	Description	Value
width	Width of the map	48
height	Height of the map	27
max_num_cities	Max number of cities to build	5
max_rails_between_cities	Max number of rails connecting to a city	2
max_rails_in_city	Number of parallel tracks in the city	3
n_episodes	Number of episode per each training run	1000
max_steps	Maximum number of allowed steps per episode	250
steps_to_train	Number of steps between each training pass	8
min_memories	Minimum amount of memories (before training)	10000
Random seeds	To allow reproducible experiments	42

Table 4.1: Environment parameters and other common parameters adopted during our experiments.

without the worry of saving plots of our progresses. To this end, we found in Weights & Biases an effective and very intuitive tool to monitor Machine Learning projects. Weights & Biases is also used by OpenAI and other leading companies, it supports many platforms and can be integrated rapidly in any project; for instance, we synced our WandB profile with our Flatland training script by means of the Python API and allowed us to start immediately logging every metric for comparing results. Additionally to a rich customizable interface to plot graph it also allows to group and filter every training session by its hyperparameters configuration.

Metrics and environment

To compare different runs, we consider two metrics, namely the percentage of trains arriving at their target station and the percentage of trains that get stuck in a deadlock; both metrics are smoothed using a running average with period 100. The number of agents is always either 1 or 3 (to address both the single as well as the multi agent situation), whereas the environment and

other common parameters that didn't change across different runs can be seen in Table 4.1.

4.2 Baseline

The first runs were focused on defining a good baseline, for the single agent as well as for the multi agent scenarios, that we could use later to compare the modification we introduced. In particular, we enabled both the Double DQN as well as the Dueling architecture variants, and we tried to see the impact of using ER against PER: the other hyperparameters are left at their default values (see Tables A.1, A.2, A.3 in Appendix A), and will be finetuned later. The results of this first runs are shown in Figure 4.1 for the arrival score and in Figure 4.2 for the deadlock scores (this last obviously only for the multi agent case).

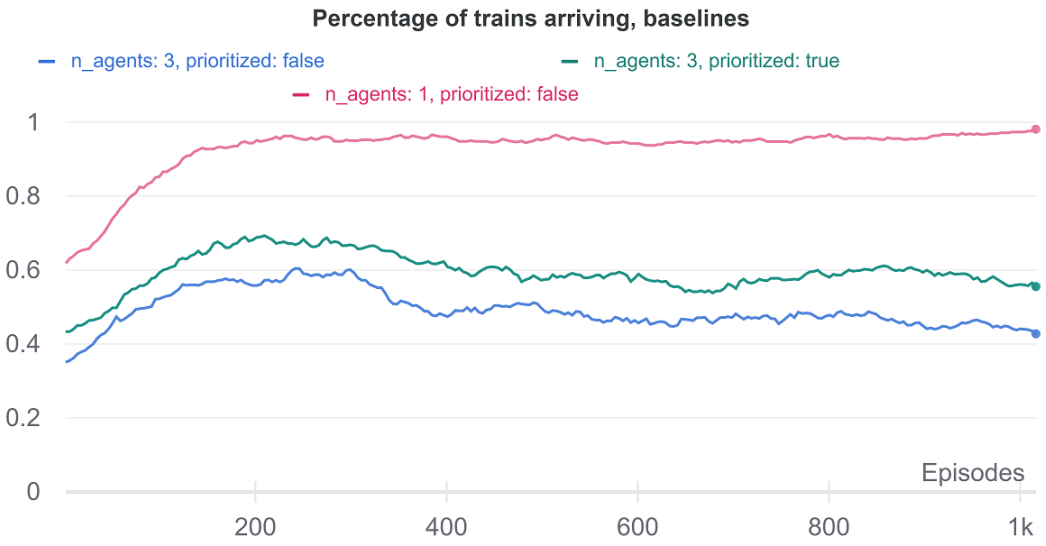

Figure 4.1: Percentage of trains arriving in the single agent scenario.

Indeed, the first thing that catches the eye is the stunning performance in the single agent case, which gets to almost 100% arrivals over 1000 episodes, without even the need of using PER (we didn't include it, given the very good result achieved anyway). This is probably due to the fact that this scenario is overly simplified - in truth we wouldn't even need RL to solve it, as it would

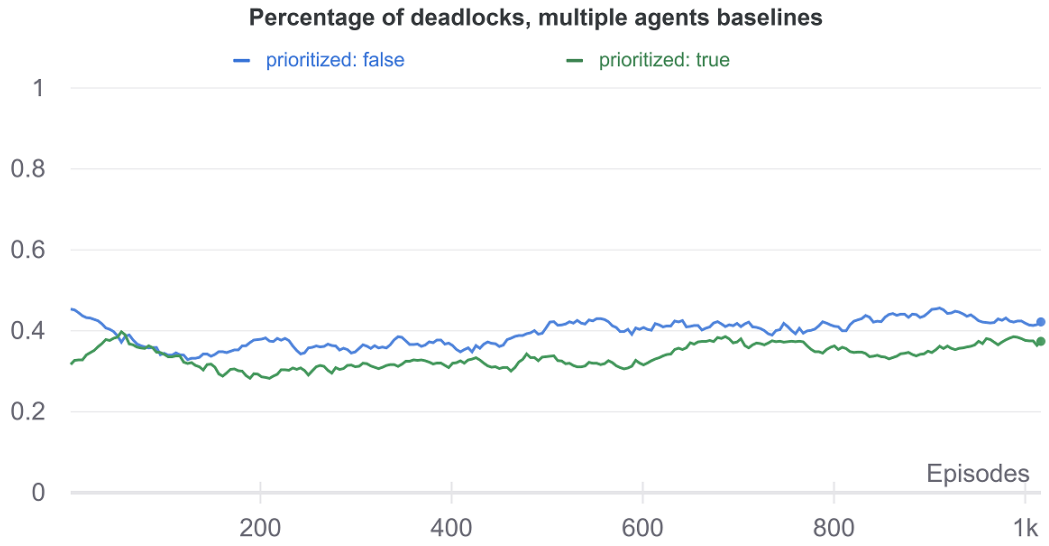

Figure 4.2: Baseline performance: percentage of trains ending in a deadlock. The curves are averaged over different runs.

suffice to use any algorithm for shortest path computation in a graph and we would have a valid (actually, a perfect) agent in such a scenario. Thus, from now on we focus only on the more complex scenario involving three trains rather than one.

Concerning the multi agent scenario, it appears that PER slightly helps, especially when considering the arrivals score - a smaller difference between ER and PER can be observed in the deadlock score. In the following tests, we will mostly keep separated the two categories - agents using PER and ones using ER.

4.3 Reward Shaping

The next runs involved testing the shaped reward signal that was introduced in Section 3.2. We consider two possible configurations of the shaped signal, which can be seen in Table 4.2. The results can be seen in Figure 4.3 (arrivals) and Figure 4.4 (deadlocks); to avoid a chaotic graph, we include only the results for agents with PER enabled.

Parameter	Configuration A	Configuration B
α	1	1
β	5	5
$p^i_{dead}(t)$	-5	-3
$p^i_{stand}(t)$	-1	0
θ	0	0.8
$p^i_{stop}(t)$	-0.2	-0.1
$p^i_{start}(t)$	-0.5	-0.2

Table 4.2: Configurations used to test the shaped reward signal.

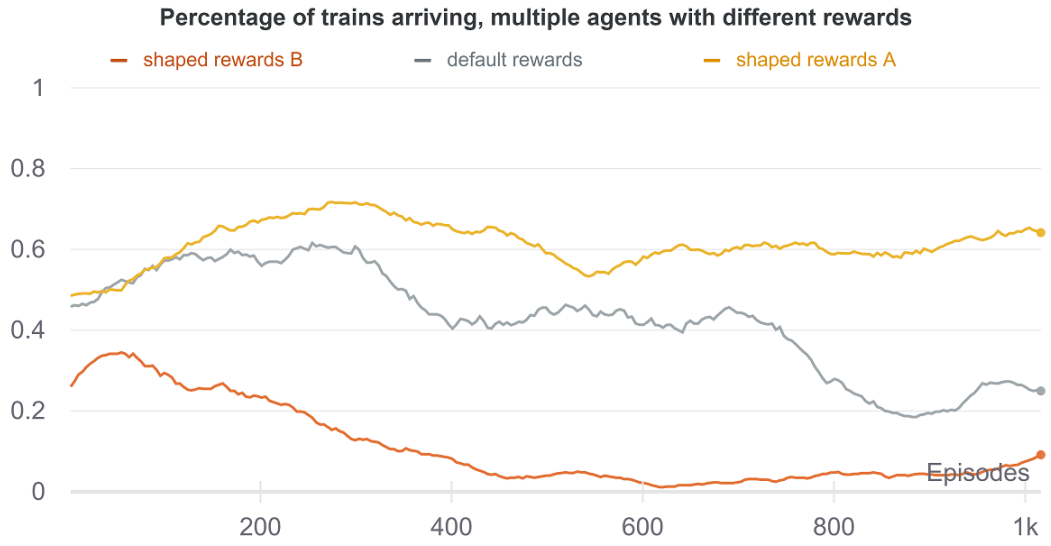

Figure 4.3: Percentage of trains arriving in the 3-agents scenario, with different reward signals. The curves are averaged over different runs.

From these results, it looks pretty clear that the shaped signal can help as well as hurt a lot, performance-wise. In particular, configuration A appears to result in a more stable training, as well as in a better performing agent in terms of percentage of arrivals; on the other hand, configuration B leads to a terrible agent with an arrival score nearing 0% after about 450 episodes. Interestingly, agents trained with configuration B are still capable of learning (although slower) to avoid deadlocks almost as much as agents trained on the other two configurations (A and default), as Figure 4.4 shows. Finally, rewards

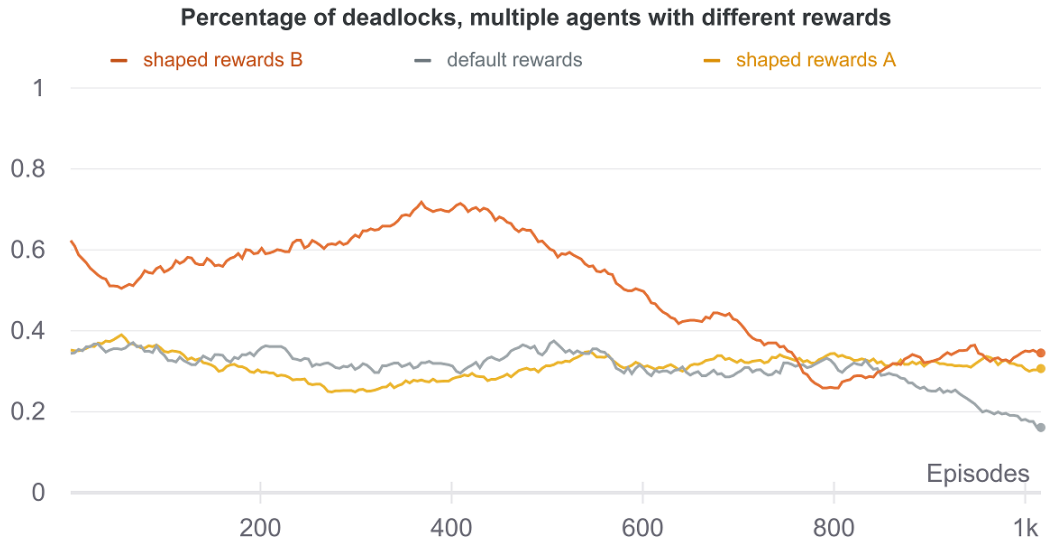

Figure 4.4: Percentage of deadlocks in the 3-agents scenario, with different reward signals. The curves are averaged over different runs.

configuration A and the default configuration are pretty similar in terms of percentage of deadlocks. Thus, from now on we will consider configuration B as the "default" configuration for the Flatland environment that we use for the subsequent tests and the baseline tests.

4.4 Iterative training

The next runs involved testing the iterative training procedure explained in section 3.3. The results can be seen in Figure 4.5 (arrivals) and Figure 4.6 (deadlocks). By monitoring also the percentage of deadlocks, it is possible to notice that the different trainings we run, on average performed better with the preliminar iterative procedure. The peak performances were reached by runs with the iterative procedure with a mean score of 80% of train arrivals and 16% of deadlocks. In this context, PER doesn't seem to have much of an impact in terms of absolute performance; the only difference, which is more noticeable when considering the arrival scores, is that training a little more stable.

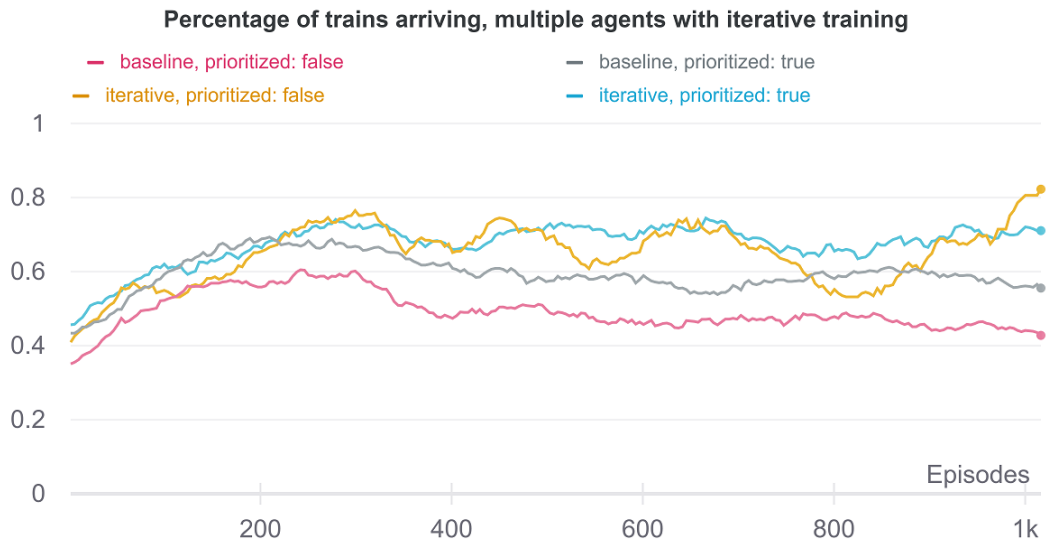

Figure 4.5: Percentage of trains arriving in the 3-agents scenario, using iterative training. The curves are averaged over different runs.

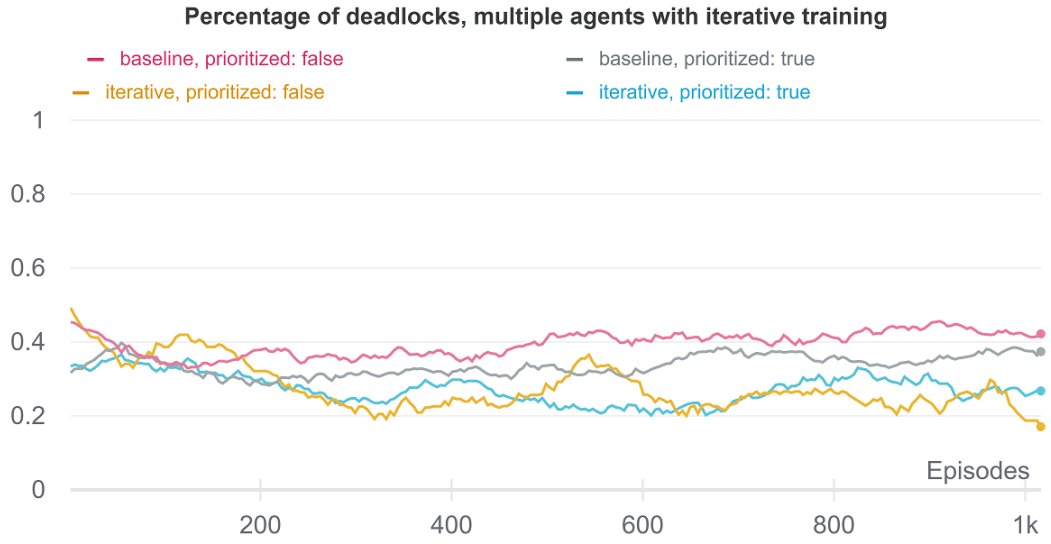

Figure 4.6: Percentage of deadlocks in the 3-agents scenario, using iterative training. The curves are averaged over different runs.

4.5 Custom observation

Due to the fact that the custom observation shape depends on the number of agents that are present in the environment, it hasn't been possible to combine the custom observation with the iterative procedure because of the

different shapes of the input.

As outlined in section 3.4, by adding new information to the network's input hopefully the agent should perform better due to a better understanding of the "global" situation.

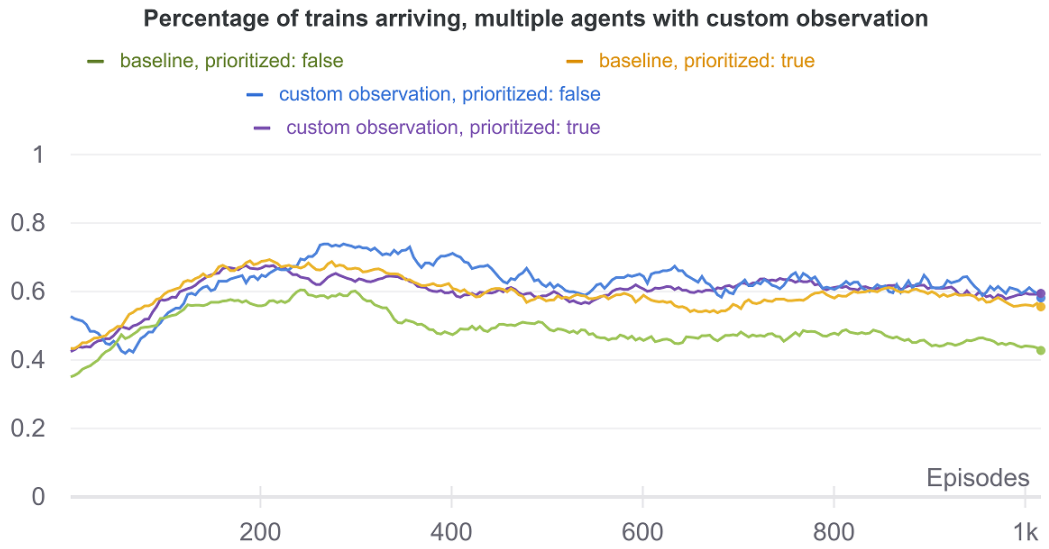

Figure 4.7: Percentage of trains arriving in the 3-agents scenario, using the custom observation during training. The curves are averaged over different runs.

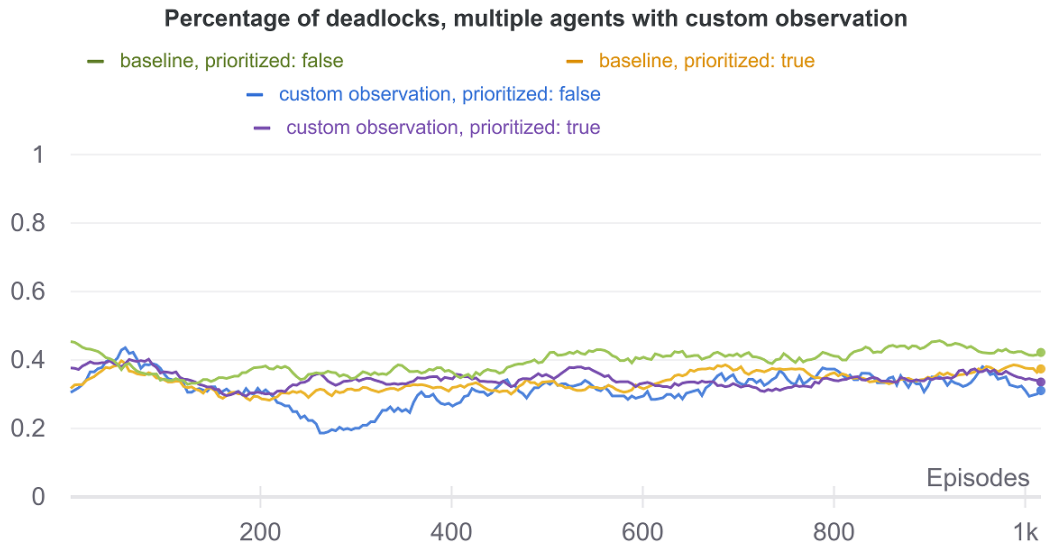

Figure 4.8: Percentage of deadlocks in the 3-agents scenario, using the custom observation during training. The curves are averaged over different runs.

By looking at Figure 4.7 and Figure 4.8, it appears that runs with the custom observation with the ER led to better results, meanwhile the runs with the custom observation with PER seem to be as good as the baseline with PER; this confirms the hypothesis previously stated about the more information in input the better the results.

The peak performances were reached by runs with the custom observation with a mean score of 80% of train arrivals and 8% of deadlocks.

4.6 Hyperparameters tuning

During the different phases of the experiments, some hyperparameters were tuned in order to try to reach the best performances.

One of the most important hyperparameters in a DL context is indeed the learning rate: different values were tested, ranging from 0.00045 to 0.001.

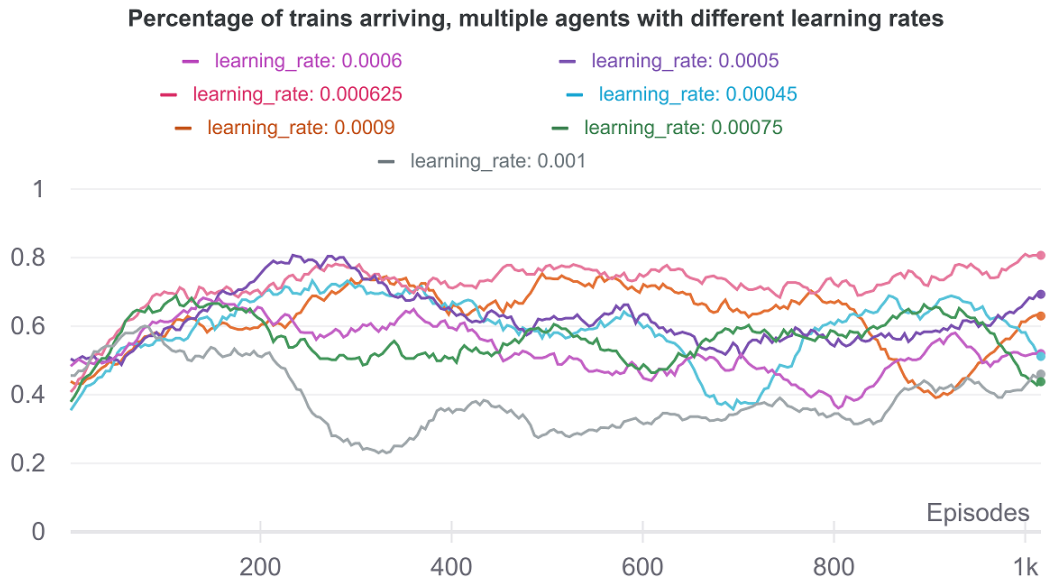

Figure 4.9: Percentage of trains arriving in the 3-agents scenario, with different learning rate.

By observing Figure 4.9 and Figure 4.10, it is noticeable that the best learning rates were 0.000625 and 0.0005.

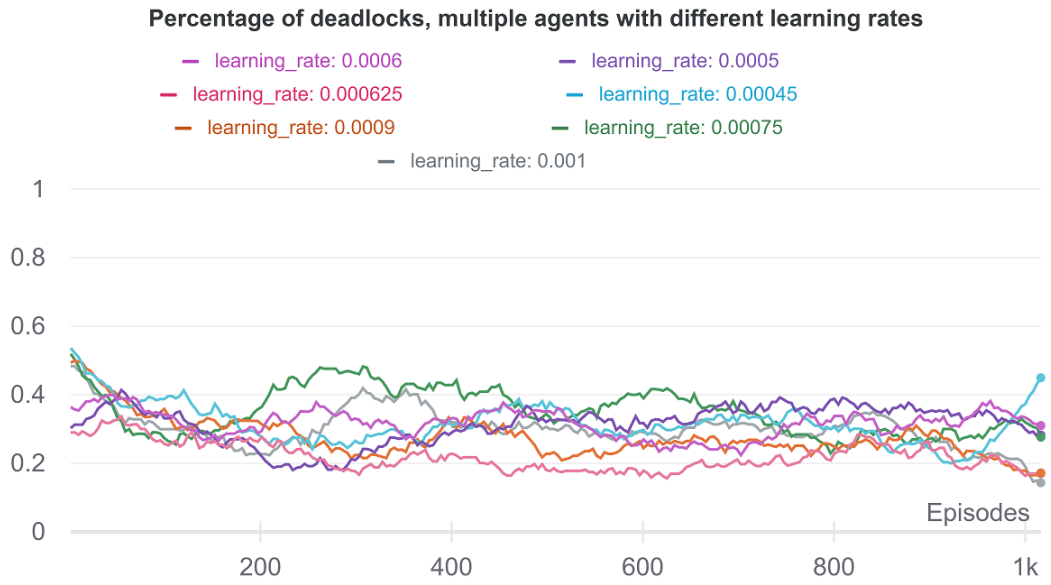

Figure 4.10: Percentage of deadlocks in the 3-agents scenario, with different learning rate.

Chapter 5

Conclusion

In this work, we tested different solutions to the Flatland challenge, leveraging both well-established techniques as well as new and creative ideas.

First, we started by reimplementing from scratch the classical DQN algorithm and some of its most famous variants, like Double DQN, Dueling architectures and Prioritized Experience Replay; after that, a more creative phase involved the design and implementation of a different training procedure and of a custom observation. The idea of our custom observation showed that the enhancement of the Tree observation with some additional, environmental information can indeed improve the performances. Reward shaping also played a big part in obtaining better results.

The results we've obtained are still lower than the ones seen on the Flatland's leaderboards, but they show that through some hypeparameter tuning (and especially a more powerful training environment) we could reach good scores.

Flatland, and more generally MARL, is a very hard problem, so that strong solutions are needed in order to have an agent that can generalize well over a wide range of environments. Research on this field is growing every year, and we believe that the combination and integration of different techniques could really prove useful in tackling hard problems such as RL and MARL problems.

Appendix A

Default hyperparameters

Hyperparameter	**Description**	**Value**
activation	Activation function used by the network	relu
conv_layer_params	Hyperparameters of convolutional layers (if any)	None
fc_layer_params	Hidden units of fully connected layers	(128, 128, 64)
dropout_params	Dropout rates (if any, must be same length as fc_layer_params)	None
dueling	If True, enable dueling architecture	True

Table A.1: Network hyperparameters with default values.

Hyperparameter	**Description**	**Value**
alpha	(PER only) Alpha parameter	0.5
beta	(PER only) Beta parameter for importance sampling	0
size_short	Size of short term memory	5000
size_long	Size of long term memory	50000

Table A.2: Buffer hyperparameters with default values.

Hyperparameter	**Description**	**Value**
batch_size	Batch size to use during training	128
ddqn	If True, enable Double DQN loss	True
gamma	Discount rate	0.99
iterative	If True, enables multi-agent iterative training	False
learning_rate	Learning rate	0.0005
multiple_obs	If True, use custom observation	False
prioritized	If True, use PER	True
target_update_period	Number of steps before updating target net	100
tau	Controls soft vs hard target updates	1

Table A.3: Agent hyperparameters with default values.

Bibliography

[1] Jing-Quan Li, Pitu B Mirchandani, and Denis Borenstein. The vehicle rescheduling problem: Model and algorithms. *Networks: An International Journal*, 50(3):211–229, 2007.

[2] Flatland environment.

[3] Sharada Mohanty, Erik Nygren, Florian Laurent, Manuel Schneider, Christian Scheller, Nilabha Bhattacharya, Jeremy Watson, Adrian Egli, Christian Eichenberger, Christian Baumberger, et al. Flatland-rl: Multi-agent reinforcement learning on trains. *arXiv preprint arXiv:2012.05893*, 2020.

[4] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[5] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent reinforcement learning: An overview. *Innovations in multi-agent systems and applications-1*, pages 183–221, 2010.

[6] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[8] Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23:2613–2621, 2010.

[9] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[10] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[11] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[12] Jürgen Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In *Proc. of the international conference on simulation of adaptive behavior: From animals to animats*, pages 222–227, 1991.

[13] Marek Grzes. Reward shaping in episodic reinforcement learning. 2017.

[14] Adam Daniel Laud. *Theory and application of reward shaping in reinforcement learning*. University of Illinois at Urbana-Champaign, 2004.

[15] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[16] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E Taylor. A survey and critique of multiagent deep reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 33(6):750–797, 2019.

[17] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.