TECHNISCHE
UNIVERSITÄT
DARMSTADT

Department of Mathematics

Master Thesis

Evaluating Planning-based Machine Learning Algorithms for Scheduling Railway Operations

Lorenz Leichthammer

31.03.2022

Supervisor: Johannes Czech
First reviewer: Prof. Dr. Kristian Kersting
Second reviewer: Prof. Dr. Marc Pfetsch

Abstract

Delays happen inevitably during railway operation. In a busy network even delays of a few seconds can spiral out of proportion and cause disruptions on a large scale, if they aren't handled appropriately. Currently, the conflicts arising due to delayed trains are handled by human dispatchers operating on local sections of the railway grid. Previous research to improve the dispatchers' decisions primarily focused on computing the appropriate conflict resolution as a solution of an optimisation problem. In principle Machine Learning methods can also improve the quality of conflict resolution. We use Deep Q-learning on simplified railway grids and compare trained networks to heuristic baselines. The results indicate that it is difficult to improve conflict resolution decisions even over simple baseline solutions, but could be achieved by carefully tuning the training environment. However, bad scaling may prevent an application on a large scale.

Keywords: Conflict Resolution, Reinforcement Learning, Deep Q-Learning, Flatland

Kurzfassung

Im Bahnbetrieb lassen sich Verspätungen nicht vollständig vermeiden. In Schienennetzen mit hoher Auslastung können selbst Verspätungen von wenigen Sekunden eine Reihe von Konflikten auslösen, die große Störungen im Zeitplan nach sich ziehen, wenn nicht adäquat auf sie reagiert wird. Derzeit bestimmen menschliche Fahrdienstleiter, wie auf solch ungeplante Konflikte in lokalen Abschnitten des Schienennetzes zu reagieren ist. Bisherige wissenschaftliche Arbeiten, die diese Konfliktlösungen verbessern wollten, haben sich hauptsächlich damit beschäftigt, die Entscheidungsmöglichkeiten als Optimierungsproblem zu formulieren und zu lösen. Im Prinzip können auch Machine Learning Methoden dazu beitragen die Qualität der Konfliktlösungen zu verbessern. Wir haben Deep Q-Learning auf vereinfachten Schienennetzen eingesetzt und die trainierten tiefen Netze mit heuristischen Basislösungen verglichen. Die Ergebnisse zeigen, dass selbst sehr simple Basislösungen schwierig zu übertreffen sind, aber dies mit Hilfe von sorgfältig gefertigten Trainingsumgebungen möglich sein könnte. Die schlechte Skalierbarkeit ist jedoch eine Herausforderung für den Einsatz auf größeren Umgebungen.

Schlüsselwörter: Conflict Resolution, Reinforcement Learning, Deep Q-Learning, Flatland

Contents

List of Figures

List of Tables

Introduction

Whenever delays are measured during railway operation, local dispatchers become active and coordinate the affected train traffic in order to minimise any knock-on effects that the delay might have on the timetable. This coordination includes the resolution of conflicts, which occur due to the shift of the delayed train's space-time trajectory. Assistance to dispatchers for their decision making is currently given either by contingency plans that were developed beforehand or by real-time solutions of optimisation problems, refer to [CHK+14] or [FYY15] for an overview of the developed approaches.

The increasing number of applications of Machine Learning methods gives rise to the question, whether they can also be employed in the context of railway operation to improve its quality. Out of the many different aims for the different target groups, see [HP14] (page 275) for an overview, we choose the passengers' dissatisfaction as an optimisation criterion to make train traffic a reliable and attractive mode of transportation for passengers.

Since during railway operation no labelled data is available, the preferred Machine Learning approaches have been Reinforcement Learning methods, which need to teach themselves to select the best available action in every situation. So far they have been employed mostly on the Multi-agent Pathfinding problem, in which multiple agents have to coordinate themselves step by step in order to all reach their targets without blocking each other. Advances in this direction have been additionally encouraged by the Flatland challenge[1].

The goal of this thesis was to combine algorithms from Reinforcement Learning with planning algorithms to solve conflicts in train schedules that arise due to unforeseen delays and to compare the results of our new approaches to those, which are currently used in practice. In the end we achieved a comparison between trained deep networks and heuristic baseline solutions, which are all tasked to choose an appropriate conflict resolution from a limited set of options, whenever conflicts arise

[1] https://flatland.aicrowd.com/intro.html (accessed on 16.03.2022)

in railway operation simulation.

To simulate railway operation we first had to find an environment that allowed us to generate fixed and random railway grids. Since implementing a new simulator from scratch wasn't feasible in the limited time period of this thesis, we relied on the existing one of the Flatland challenge.

Chapter 1 presents papers that are closely related to this thesis and gives more details about the Flatland challenge. In chapter 2 we discuss the necessary steps that had to be taken prior to the simulations and in chapter 3 we present their implementation and the different environments that we were using. Chapter 4 is dedicated to conflict detection and conflict resolution, which are two central aspects of this thesis. Chapter 5 gives a brief summary of Reinforcement Learning and its algorithms that we used. In chapter 6 we present the results of our simulations and in chapter 7 we conclude the thesis and give an outlook for future work.

All implementation can be found by accessing https://gitlab.com/lcleicht/flatland, which is a fork project of https://gitlab.aicrowd.com/flatland/flatland/-/tree/master/. A list of all files that were changed or newly written is given in the appendix A.1.

Acknowledgements

First and foremost I would like to thank my supervisor Johannes Czech for introducing me to the topic of Machine Learning in railway scheduling and for his guidance through the time that I could work on this thesis. During our many meetings he gave me invaluable ideas and insights to steer the course of this project in the appropriate direction even in the most troublesome times. I would like to extend my thanks to the whole team at the *Institut für Bahnsysteme und Bahntechnik* for coming up with the idea of such a cooperational project and especially to Arturo Crespo Materna, who acted as a second supervisor and provided endless information about anything railway related. I would like to thank my professors Prof. Dr. Kristian Kersting and Prof. Dr. Marc Pfetsch for agreeing to review this thesis. My gratitude for proofreading goes to Anny Zheng, Blendi Shala, Hilal Aksoy, Justus Pritzl, Lina Briesemeister, Timon Dorschner and once again Arturo Crespo Materna.

Chapter 1

Related Work

We start this chapter with a summary of a paper about Deep Reinforcement Learning for single track train scheduling and discuss the similarities and differences to this thesis. We continue with an introduction to the Flatland challenge as it is a good model environment to experiment with scheduling and rescheduling algorithms and provides easy access to discover the potential of Machine Learning algorithms to enhance railway operation. We will discuss Flatland's implicit assumptions, which may not always hold in real train traffic and need to be changed for the purpose of this thesis. Additionally, we will also compare different possibilities to evaluate the quality of railway operation.

1.1 Reinforcement Learning for train scheduling

The paper that arguably has the most similarities with this thesis regarding both the desired goal and the applied methods to achieve this goal is *Solving the single track train scheduling problem via Deep Reinforcement Learning* by Grani et al. [AGL20]. In it the authors are tackling the train dispatching problem, which has the goal to minimise the weighted delay of the entire network by using Deep Reinforcement Learning to route the trains. Minimising the delay is very similar to minimising the passengers' dissatisfaction, which we will discuss in more detail in the following sections of this chapter.

However, there are several differences to this thesis. First of all they were training and testing their algorithms on a fixed single track railway network, while we will use both fixed and randomly generated Flatland environments. Second the train dispatching problem is closely related to the Multi-agent Pathfinding problem in the sense that their goal was to choose singular moves for all trains at all time steps.

For this purpose they were comparing two approaches (centralised and decentralised Deep Q-learning) to linear Q-learning. We on the other hand are more concerned with conflict detection and conflict resolution in the plans of whole episodes. This means that we have got an initial plan for each train specifying exactly where it should be at any time step and only if something goes wrong, we would like to adjust that plan. We also choose Deep Q-learning as our Machine Learning algorithm, but we compare our approaches to simulations of realistic train traffic abiding to the decision rules that dispatchers apply in practice and other simple baseline solutions. Third the features or observations and the training parameters are chosen differently. In the paper the authors chose the features to be for every part of the network its status, the number of trains in it, the priority of the trains, their directions, a length check and the number of parallel resources. As training parameters they chose the number of trains, their initial positions, their initial directions, their priorities and their length. Since our focus lies on conflict resolution, we use for all resolution options the trajectories of the trains and information about possible follow-up conflicts as features. The training parameters consist in our case of the size of the environment, the number of trains, their starting times, positions and directions, their targets, the number of cities, their sizes and the malfunction distribution.

1.2 The Flatland challenge

To simulate the railway rescheduling problem we need an infrastructure model, an operating program, a prognosis module to model the trains' driving dynamics, a stochastic delay generator and the time reserves in the timetable, see [CM20] (page 8). The Flatland challenge[1] offers most of these functionalities as an open source project, which makes it suitable to be used as a simulator for this thesis.

The Flatland challenge is an open competition, in which everybody can participate and send in their solution to compete for the first place by gaining the most points. The setting is a 2D-railway grid with stations and switches, on which trains should find their way from their origins to their destinations. Points are awarded for trains reaching their destinations and for completing an environment, which means that every train in the scenario has reached their destination. These environments become successively more difficult as the grids become larger and the number of trains becomes greater. To make sure that every solution has equal chances a time

[1] https://flatland.aicrowd.com/intro.html (accessed on 16.03.2022)

limit is set, after which the computation is interrupted and no more points can be gained. At the time of writing the Flatland challenge is in its third edition, so we will discuss the best approaches that have been brought forward so far and in particular focus on the models they used.

1.2.1 Operations Research

Even though one of the goals of the Flatland challenge was to promote advances in Reinforcement Learning (RL), it turns out that the RL methods currently can't compete with Operations Research (OR) or mixed methods, see [LSS+21] (page 16) or the results of last year's Flatland challenge[2]. Therefore, in order not to give up on RL methods altogether, the organisers split the solutions in two categories, "RL methods" and "Non-RL methods". We will first take a look at the Non-RL methods, where even the sixth place accumulated more points than the best RL method. The best OR method is described in the paper by Laurent et al. about the Flatland 2020 challenge [LSS+21](pages 8-10). The underlying idea was to use a Multi-Agent Pathfinding algorithm, at which we are going to take a closer look.

Multi-Agent Pathfinding

Following the notation of Stern et al. [SSF+19] (pages 151-153) the classical Multi-agent Pathfinding (MAPF) problem receives as an input a tuple (G, s, t), where $G = (V, E)$ is an undirected graph, s is a vector of dimension k that contains the source vertices of all agents, t is also a vector of the dimension k containing the target vertices and k is the number of agents. The goal is to find a sequence of actions that the agents can execute over discretised time steps thereby moving each agent from its source vertex along the edges of the graph to its target vertex while avoiding collisions with the other agents. Actions can be to move the agent from one vertex along an edge to a neighbouring vertex or to remain stationary. Usually the objective function of an MAPF problem is the makespan, the time until all agents have reached their target, which is to be minimised.

In Flatland however this isn't true anymore. Since there is a time limit, the goal is to bring as many agents as possible to their target location in the given time limit. Other specifics of the Flatland environment, which don't apply to the classical MAPF problem, are that malfunctions can happen, thus locking agents in place for a certain amount of time. Also agents can be held back from entering

[2] https://www.aicrowd.com/challenges/flatland/leaderboards (accessed on 16.03.2022)

the grid, which may be sensible, if it is already crowded and agents vanish at their target vertex, so they don't block traffic passing the location later. While vanishing agents at their target and the possibility to hold back agents make the problem easier to handle, there are also restrictions of the Flatland environment that aren't present in the classical MAPF like only being able to move forward or make a turn at a switch. Reversing isn't possible in Flatland.

The winner of the Flatland 2020 challenge

The best solution of the Flatland 2020 challenge is called "An_Old_Driver" and was developed by Li, Chen, Zheng and Chan [LCZ+21]. It used an MAPF algorithm to calculate collision-free paths by assigning priorities to the trains and scheduling them by descending priority while avoiding conflicts with already scheduled trains. Malfunctions weren't considered in the scheduling and only handled during execution to solve the arising conflicts. Their conflict resolution was done by partial replanning, which means that when a train had a malfunction, the order, in which trains pass through intersections, was checked and in case of need adjusted for all intersections in the remaining path to the malfunctioning train's target.

1.2.2 Reinforcement Learning

There are many design choices in Reinforcement Learning, refer to [SB18] or chapter 5 for more details, and one of the most important is to decide, which features the agent observes and uses to compute its actions. It is necessary to select features that contain information, but the simple solution of using every observable feature may not be optimal, because features without much information add noise to the model and worsen the performance. Therefore, the best performing solutions of the Flatland 2020 challenge in the Reinforcement Learning category, also described in [LSS+21] (pages 10-14), only used some carefully selected features usually coupled with an observation tree of depth between 1 and 3 to scout the grid ahead of their agents.

The reward function is another part, which heavily impacts the agents' behaviour depending on its terms and corresponding coefficients. The best performing RL solutions all penalised deadlocks and rewarded agents reaching their target. The two best RL solutions approached the reward function in very different ways: one using positive rewards as an incentive for agents to move closer to their target, the other using negative rewards to punish unwanted behaviour such as disrespecting priority, not following the shortest path or unnecessary stopping.

To exploit the Flatland environment and the characteristics of the challenge the best performing RL solutions all used some additional functionalities. The most widely used was a departure schedule, which we already mentioned as holding back of trains in the section about MAPF. This was absolutely necessary to achieve high scores, because during evaluation the number of trains was rapidly increased for each environment, which led to crowded environments, if no trains were held back. Other functionalities that seemed to help improve performance were communication between the trains and assigning priorities to the trains to better solve conflicts.

The training is where there were the most differences, since three of the four best performing RL solutions used different methods. The best solution used Proximal Policy Optimization [SWD+17] and the second best solution used centralised training of an Ape-X model [HQB+18]. The fourth best solution used a multi-agent decentralised framework called PRIMAL [SKS+19] and learned via the A3C learning algorithm [MBM+16].

Towards conflict resolution

The goal of this thesis isn't to code a Reinforcement Learning agent that does well in the Flatland challenge. In fact, we intentionally imposed rules, which prevent good performance in the challenge, in order to have more realistic railway operation in Flatland like keeping a minimum headway between trains or changing the process in charge of generating the environment. Instead the Reinforcement Learning agent should select appropriate conflict resolution, when such conflicts arise during railway operation, for which Flatland is a convenient simulator.

1.3 The passengers' dissatisfaction problem

There are many different aspects that can be used to measure the quality of a running railway operation, see [HP14] (page 275). In the Flatland challenge the only indicator has been the number of trains reaching their target in a given time and only now in its third edition we saw the introduction of schedules and thus the possibility to reward punctuality and punish trains arriving late at their target. However, this isn't the only objective one could have in mind when optimising railway operation, so research has been done on different optimisation problems with different objective functions depending on the focus of the work at hand. For example, Schöbel investigated the delay management problem in [Sch01] (pages 1-2). The objective of the delay management problem is to minimise the total delay

experienced by all passengers, which needs to be achieved by deciding whether to keep or break connections in the scheduled timetable, when the first vehicle is delayed. Törnquist and Persson formulated a mixed integer linear program in [TP05] (page 4) with two different objective functions. One considered only the total delay of all trains at their end station, the other minimised the delay of each train at every station multiplied by a factor indicating how many people would get off plus the number of missed connections. Corman et al. even proposed a whole spectrum of indicators in [CDH10] (pages 9-10), which consists of

- travel time: the time that passengers spend on a train,
- waiting time: the time that passengers spend at a station while waiting for a train,
- generalised travel time: a weighted sum of travel time and waiting time, where waiting time carries a greater weight, because passengers perceive it as more unpleasant,
- minimum cycle time: the time needed to dispatch all scheduled trains,
- track occupancy: the ratio between the minimal time required to run all trains and the time needed for the actual operation and
- output delay: the train delay at its exit from the grid.

In this thesis we will adopt the focus of Sato et al. in [SKT18] (pages 8-12) on the passengers' dissatisfaction problem in order to cater to the passengers' demands and make train traffic a reliable and attractive mode of transportation. The objective is to minimise the total passengers' dissatisfaction that consists of

- arrival delay: the train delay upon its arrival,
- departure delay: the train delay at its departure,
- waiting time: the time intervals between two consecutive trains,
- dwell time: the time that a train spends at a station,
- running time: the time that a train spends in between stations and
- connection: if a connecting train can't be reached.

Since in Reinforcement Learning the objective is to maximise the reward function, we modelled our reward function to include the different aspects of passengers' dissatisfaction.

Chapter 2

Theoretical aspects of railway operation

Let us start with the theoretical aspects of railway operation and go step by step from an empty railway grid to a system with a given train schedule that stochastically experiences delays, which in turn create conflicts that have to be resolved. In this chapter we will discuss each of these single steps in general and in the next chapter we will discuss how these steps were implemented in the specific case of Flatland.

2.1 A realistic railway grid

In the beginning of this thesis we had to find a simulator to run the training and testing of different Machine Learning algorithms on both random and fixed railway grids. One of the fixed grids is a realistic model of a small railway grid in full detail. The grid may seem very simple, but even these small numbers of stations and junctions already need 11'340 lines of code in an xml-document to account for all signals, platforms, switches etc. that are required for safe and effective use of the infrastructure. The grid consists of three stations and two junctions in between as shown in figure 2.1 below.

The three stations are abbreviated ZDMA (left), ZDW (middle) and ZDK (right) and the two junctions are called ZDF (left) and ZDBU (right). In figure 2.2 we zoom in on the middle station ZDW, so that we can see more details than just the overall structure.

According to [CM20] (page 8) to simulate the railway rescheduling problem we need an infrastructure model, an operating program, a prognosis module to model

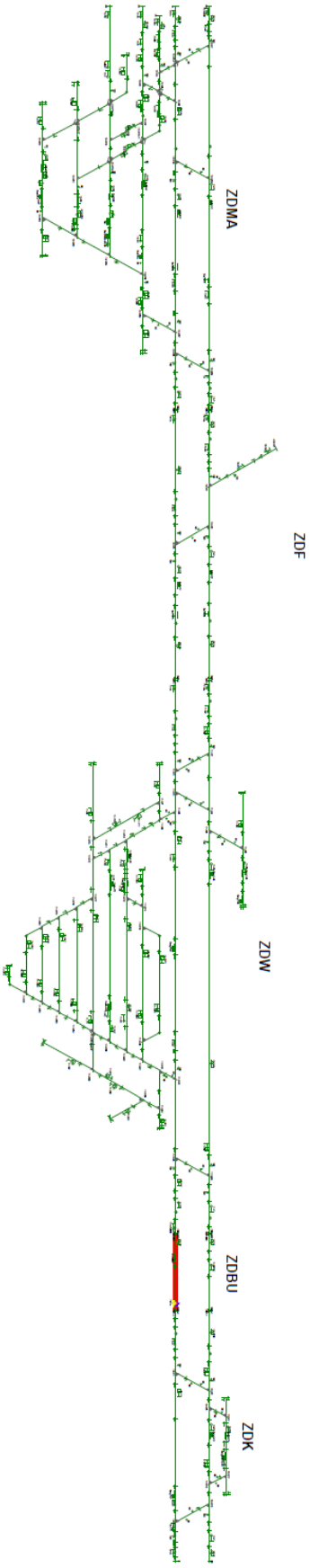


Figure 2.1: The full railway grid

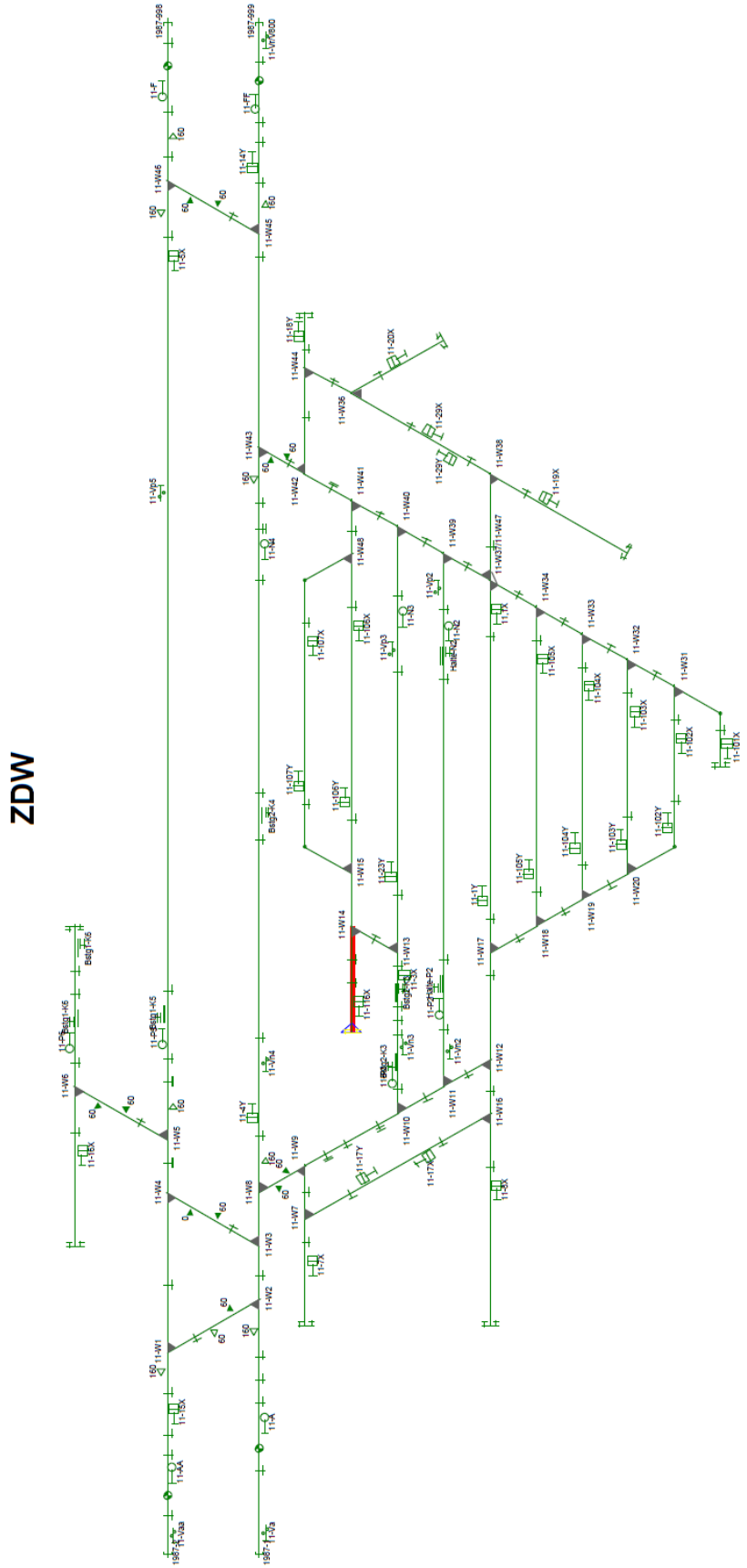


Figure 2.2: The middle station ZDW

the trains' driving dynamics, a stochastic delay generator and the time reserves in the timetable. Converting the infrastructure model from figure 2.1 into a tractable format doesn't pose much of a problem, since there exist standard parsers to transform xml-documents into Python element trees. Element trees are particularly suited to represent the infrastructure, because they preserve the hierarchical order of the xml-document.

The other functionalities however are much more difficult and time consuming to implement. Therefore we decided not to spend the whole duration of this thesis on the development of a new realistic simulator and instead use the existing one of the Flatland challenge, even though it is not realistic in the sense that there are no signals, there is no minimum headway between trains, no blocking of switches while its direction is changed and many more restrictions that we have in real life. The advantages are that Flatland offers a full operating program, where infrastructure models can be easily created and train driving dynamics are modelled in a sufficiently realistic way.

2.2 A realistic schedule

Before we can gather information about time reserves in the timetable, we first must create a timetable. For our model grid from figure 2.1 there is a realistic schedule created by the team of the *Institut für Bahnsysteme und Bahntechnik* at the TU Darmstadt. The schedule contains 50 trains and spans a time period of two hours. These 50 trains contain 24 slow regional trains (S-trains) and 26 fast regional or long distance trains, which are distributed over eight lines. An overview of the origins and destinations as well as the departure times of all trains is given in table 2.1.

2.3 Disturbances

To make railway operation in the Flatland challenge more realistic a stochastic delay generator was introduced already in the first version to simulate disturbances and delays, which happen in daily railway operation in the real world. In this section we are going to discuss how disturbances and delays can be modelled and what assumptions are made using these models.

Following the notation from Cui, Martin and Zhao [CMZ16] (pages 2-4) we define a *disturbance* as a stochastic event that a time extension occurs for a train during railway operation. However, it isn't a *delay* until the time extension is measured at

Train lines of the custom schedule						
Line	From	To	Departures	From	To	Departures
S1	ZDW	ZDK	08:00	ZDK	ZDW	08:00
			08:15			08:15
			08:30			08:30
			08:45			08:45
			09:00			09:00
			09:15			09:15
			09:30			09:30
			09:45			09:45
S2	ZDW	ZDM	08:10	ZDM	ZDW	08:25
			08:40			08:55
			09:10			09:25
			09:40			09:55
RB20	ZDM	ZDW	08:39	ZDW	ZDM	08:55
			09:39			09:55
RB40	ZDW	ZDK	08:21	ZDK	ZDW	08:07
			08:51			08:37
			09:21			09:07
			09:51			09:37
RE50	ZDM	ZDK	08:05	ZDK	ZDM	08:19
			09:05			09:19
IC100	ZDM	ZDK	07:53	ZDK	ZDM	07:50
			08:53			08:50
			09:53			09:50
IC300	ZDM	ZDK	08:09	ZDK	ZDM	08:23
ICE500	ZDM	ZDK	08:22	ZDK	ZDM	08:09

Table 2.1: The schedule on our model grid for a time span of two hours

a measurement point. Before that the disturbance can still be recovered and in this case not cause a delay.

The distribution of the stochastic time extensions is modelled as a negative exponential function with an additional parameter to first distinguish between trains with and without time extension. Time extensions themselves can be classified as running time extensions, dwell time extensions or departure time extensions. It should be noted that the choice of the negative exponential function is a modelling assumption and not necessarily the best function to model disturbances. Other propositions to model the disturbances have been made, most notably by Yuan [Yua06] (pages 57-70), who found lognormal and Weibull distributions to be good approximators.

In our implementation we will use a negative exponential function, because that is also the distribution that train companies like *Deutsche Bahn* use to model delays. Regarding time extensions we will consider running time extensions and departure time extensions. Dwell time extensions don't need to be considered separately, because they can be modelled as running time extensions of the time that the train spends at a platform in the station.

Chapter 3

Flatland specifics

In this chapter we are going to revisit the requirements for the railway rescheduling problem from the previous chapter and discuss their implementation in Flatland in more detail. Additionally, we are going to address some Flatland or Reinforcement Learning specific issues like controllers or reward functions.

3.1 Building the infrastructure in Flatland

We still need a tractable model of the infrastructure. In Flatland this means that we need to define the size of the rectangle, in which our grid is contained and then determine for every 2D coordinate of the rectangle, which of the 8 unique cells of the Flatland environment is located there. The different types of cells are shown in the figure below. Note that we can also mirror and rotate them.

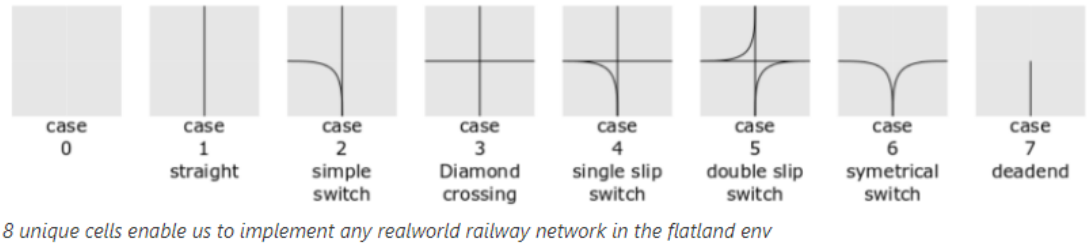


Figure 3.1: Screenshot from the Flatland website showing the different types of cells[1]

In order to create the following Flatland grid resembling our original model grid from figure 2.1, we can now fill in these cells into a 27x80 rectangle. The grid

[1] https://flatland.aicrowd.com/getting-started/env.html (accessed on 11.11.2021)

contains three cities with train stations. In each city one of the train stations is marked by a small house, which indicates that it is currently the target station of a scheduled train. Note that the grid was later changed for the experiments to 27x118 cells and longer tracks between cities to make it more realistic, but we only show here the short version so that the details don't become too small.

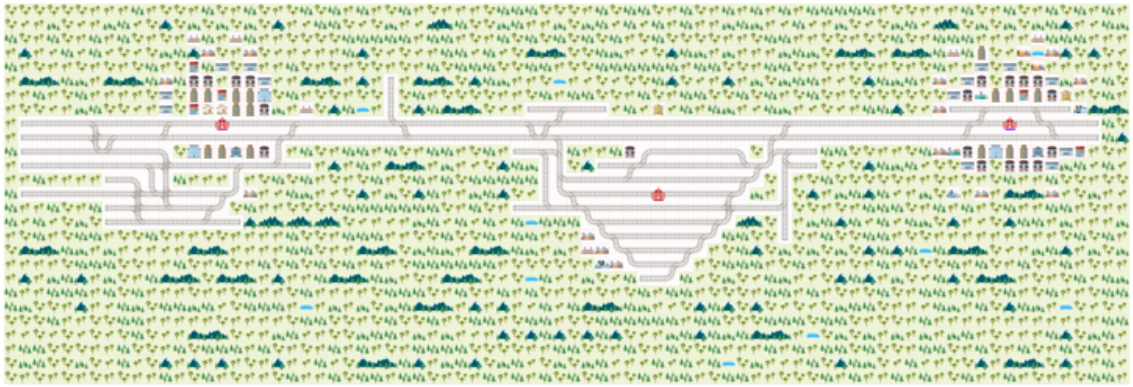


Figure 3.2: The model grid in Flatland (short version)

The code that was used to create this grid is the "make_custom_rail" function[2], which is based on the existing function "make_simple_rail_with_alternatives()" from the same file.

3.2 Reward function

A vital part of Reinforcement Learning is to define the reward function, refer to [SB18] (pages 53-55) or section 5.3 for more details about rewards and returns. The purpose of the reward function for participants of the Flatland challenge is to compare the behaviour of different agents and to evaluate the algorithms controlling them. The evaluation is supposed to be done in a way that encourages behaviour like reaching the target station or doing so in a given time horizon and punishes unwanted behaviour like stopping needlessly, taking unnecessary deviations or not even departing. The default reward function in Flatland is described by Mohanty et al. [MNL+20] (page 10) in the Flatland competition paper. The formula to compute the reward at timestep t for agent i is

$$r_i(t) = \alpha r_l^i(t) + \beta r_g(t) + r_p^i(t), \tag{3.1}$$

[2] https://gitlab.com/lcleicht/flatland/-/blob/master/flatland/utils/simple_rail.py

where α and β are parameters, whose values can be freely chosen, r_l^i is the local reward for agent i, which is 0 if the agent has reached its target and -1 otherwise, r_g is the global reward, which is 1 if all agents have reached their target and 0 otherwise and r_p^i is a penalty for the case that an illegal action is chosen, but is set to 0 by default.

From the formula we can see that a good controller for the Flatland challenge focuses on bringing all trains as quickly as possible to their target stations. While this is definitely a good idea, there is no order, in which the trains should reach their target, so sacrificing one of the earlier trains with an inconvenient route and leaving it out on the grid until all other trains have reached their target might be a legitimate strategy to maximise the reward function. In real life such a behaviour would of course lead to very poor passengers' satisfaction and also cause missed connections, which aren't considered in Flatland. Therefore we are going to modify the reward function so that higher scores are awarded to better solutions of the passengers' dissatisfaction problem.

In section 1.3 we have seen how the passengers' dissatisfaction was originally defined consisting of six parts: arrival delay, departure delay, waiting time, dwell time, running time and connection. We will consider here a simplified dissatisfaction computation, because in Flatland there are no connecting trains and no regular time intervals between trains departing from a station, unless such a dispatching routine is manually implemented. Additionally, we won't distinguish between dwell time and running time, so that we can combine it with the same penalty factor as the time between departure and arrival. We will also leave out the departure delay, since any departure delay will automatically lead to an arrival delay, because the planned arrival time is computed as earliest departure time plus driving time without malfunctions on shortest path. The arrival delay therefore effectively consists of three parts: departure delay, time spent in malfunction and delay due to waiting or rerouting to resolve conflicts.

We don't need to consider a penalty for trains that are departing early, because in Flatland trains have the state "waiting", in which they can't be put on the grid, until their earliest departure time is reached, when their state changes to "ready to depart". Summarising this paragraph we have the objective function

$$\min \; c_{tra} \cdot t_{tra} + c_{arr} \cdot \max(0, t_{arr}^a - t_{arr}^p),$$

where c_{tra}, c_{arr} are penalty coefficients for travel time and late arrival respectively, t_{arr}^p is the planned arrival time, t_{tra} is the travel time and t_{arr}^a is the actual arrival time. Therefore our reward function still has the same terms as in equation (3.1),

but only the local reward $r_l^i(t)$ is added incrementally at every time step. Note that we slightly change the local reward to be 0 not only when a train is done, but also when its departure time isn't yet reached, because in this case the train is still not allowed to be put on the grid.

We change the global reward in the sense that each train contributes a different positive reward equal to the travel time without malfunctions on its shortest path, if it has reached its target during the episode and denote it by $r_g^i(t)$. Before the episode finishes it is 0 and only at the end it is awarded once to those trains that have reached their target. By defining the global reward in this way we make the scores more comparable, because the maximum possible score is 0, if all trains leave at their earliest departure time, can use their shortest paths and have no malfunctions. Another benefit of defining the global reward this way is that we avoid a binary division between rewards of episodes, in which all trains reach their target and thus contribute positive rewards, and rewards of episodes, in which not all trains reach their target and thus no positive reward is awarded. Instead the differences in the resulting rewards become more gradual and reflect the percentage of trains that reach their target in an episode.

We use the penalty term $r_p^i(t)$ to punish trains that missed their latest arrival time. It is also 0 until the episode finishes and then only applied once to each train as the difference between its planned arrival time and its actual arrival time. The penalty is also applied, if the train hasn't reached its target by the end of the episode or if the train has never even departed. In the first case it is equal to the difference between the planned arrival time and the projected arrival time using the shortest path. In the second case the penalty is twice the travel time on the shortest path multiplied by a coefficient.

The minimum possible score is therefore the negative number of trains multiplied with the number of time steps in an episode plus the sum of the earliest departure times and negative cancellation penalties, which gives the following equation

$$\min score = -n_{trains} \cdot n_{steps} + \sum_{trains} (departure\ time + cancellation\ penalty).$$

Our implementation of the rewards are the functions "update_step_rewards" for the local rewards, "end_of_episode_update" for the global rewards and "handle_end_rewards" for the penalties, which are all functions of the class "RailEnv"[3].

[3]https://gitlab.com/lcleicht/flatland/-/blob/master/flatland/envs/rail_env.py

3.3 Baseline controllers

In Flatland the trains' actions are decided by a controller. We can therefore use the reward function defined in the previous section to compare different controllers in Flatland. Mohanty et al. suggest in their Flatland paper [MNL+20] (page 15) three controllers as a baseline against which other implemented controllers can be tested. Those three are:

- Random,
- Constant Go Forward Action and
- Shortest Path,

where the Random Controller chooses at every time step a random action between the five possible actions go forward, go left, go right, halt and do nothing. The Constant Go Forward Action Controller chooses at every time step the go forward action as the name suggests. Note that this is always a valid action, because of the way that the cells are created. The Shortest Path Controller always chooses the action, which follows the agent's shortest path to its target.

A function to find the shortest path of an agent to its target was already implemented in the Flatland source code and returns a list of waypoints. A waypoint is a tuple of a position and a direction, so that from the list of waypoints we can find all the cells the agent should pass and the direction it should have at every point to be able to make the next step.

Because the first two baseline controllers have got a low chance to find the targets of all agents and the third one is prone to end up with a deadlock of two trains somewhere on the grid, we implemented a few more baseline controllers. Each one is trying to make the railway operation either safer by avoiding deadlocks or more efficient. The controllers are:

- Right Hand Side Controller
- Block Path Controller
- Right Hand Side and Cluster Controller.

The Right Hand Side Controller reduces the chance of a deadlock by restricting train traffic in between two cities, where there are only two tracks, to the right track in the direction of travel, but there can still be deadlocks in the city areas. The Block Path Controller ensures that there can't be any deadlock, because it only lets a

train depart, if the whole driveway is clear and blocked for this particular train. A disadvantage of this controller is of course that only a few trains can drive on the grid at the same time, if they don't have intersecting driveways. The Right Hand Side and Cluster Controller is a mixture of the other two controllers. It completely avoids deadlocks by splitting the grid into two components. One component contains all cells in between two cities, where traffic on the right hand side is enforced, the other component contains all cells in the cities, which are grouped into clusters. A train can only enter a cluster, if its path doesn't require any cells of the paths of trains currently in the cluster. It is more efficient than the Block Path Controller, because a cell in a cluster is cleared as soon as the train has left it and can then immediately be traversed by the next train without waiting until the first train has reached its target.

Note that of the second three controllers only the Block Path Controller can be applied to all grids, in particular to randomly generated grids and not just our specific model grid. The other two can't be applied to other grids without adjusting the location of the rails between cities and the clusters, which encompass the switching areas. However, future projects could work on generating environments in a way that is compatible with the other two controllers.

The implementation of these six train movement controllers are different classes written in the same file[4].

When we changed the focus of the thesis towards conflict detection and conflict resolution, we had to compute the trains' trajectories from their departure locations to their targets. This means that the corresponding controller becomes very simple: it chooses at all times the action that is necessary to follow the planned trajectory. The complexity then moves from choosing the best action to ensuring that the trajectory is feasible and sensible at any point in time.

3.4 Creating schedules

The third version of Flatland saw the introduction of timetables. This means that every train gets an earliest departure time and a latest arrival time as its attributes during initialisation. By default the earliest departure time is chosen randomly and uniformly from the interval $[0,\ departure_{max}]$. The upper limit $departure_{max}$ is a heuristic approximation to the latest departure time such that the train on average still reaches its target before the end of the episode.

[4] https://gitlab.com/lcleicht/flatland/-/blob/master/flatland/envs/controllers.py

We implemented two different options to determine the trains' schedule. The first option is very simple. For our model grid from figure 3.2 we just apply the schedule from table 2.1 by equating the initial time step 0 to the time 07:50, which is the departure time of the first train, and then modelling one minute in the schedule as ten time steps in Flatland. We changed some of the departure times by up to three minutes to reduce the number of conflicts in the schedule, when the trains were following their shortest paths instead of the right hand side track.

The second option was specifically implemented for random environments. It chooses for each train the departure time as early as possible with the restriction that there must be no conflicts with already scheduled trains in the case that no malfunctions occur for any train and the departure time must always be later than that of the latest scheduled train plus a stochastic buffer time. That is to say that the initial schedule must be conflict-free.

The implementation of these two initial scheduling options are the functions "custom_timetable_generator" and "conflict_free_timetable_generator" in the same file[5].

3.5 Adding delays

In a perfect world without delays the trains could just keep following the schedule and always arrive on time at every planned stop. But as every commuter using train transport knows, the real world isn't perfect and delays happen. Therefore in order to build a more realistic model we introduce stochastic delays (see section 2.3 for the terminology).

In Flatland delays are created by default in the following way: for every agent at every time step a function is called that determines the number of steps that the agent is forced to spend stationary. If the number of steps is greater than 0, we say that the agent is in malfunction. The computation of the number of steps consists of two parts. First the probability for a malfunction to happen is computed. Since malfunctions are assumed to follow a Poisson process with parameter λ, the probability that a malfunction occurs at a given time step is exponentially distributed and can be found by computing $1-e^{-\lambda}$. In a second step a random number between 0 and 1 is generated and compared to the malfunction probability. If the random number is greater, then no malfunction occurs and the number of stationary steps is 0. However, if the random number is smaller, then a malfunction occurs and the

[5] https://gitlab.com/lcleicht/flatland/-/blob/master/flatland/envs/timetable_generators.py

number of stationary steps is generated as a random integer from an interval whose lower and upper limit are parameters of the model. For the evaluation of solutions in the second version of the Flatland challenge the lower limit was 20 and the upper limit was 50 time steps [LSS+21] (page 5).

We don't use the default malfunction model, because we would like the number of time steps to be distributed according to a negative exponential function as proposed in section 2.3. However, since in Flatland time only takes discrete values, we need to discretise the negative exponential function and also introduce a maximum malfunction duration, which we will leave at 50 time steps, to limit the computational effort. Therefore, since the cumulative distribution function of a negative exponential distribution with parameter λ is given by $F(x;\lambda) = 1 - e^{-\lambda x}$ for $x \geq 0$, we compute the probability of a malfunction duration lasting for i time steps as

$$\frac{1 - e^{-\lambda i} - (1 - e^{-\lambda(i-1)})}{1 - e^{-50\lambda}},$$

for $1 \leq i \leq 50$. The lower limit is 1, but a malfunction duration of 1 proved to be very problematic in the way that it couldn't be detected and only cancelled a single action of a train. Therefore, whenever the distribution yielded a malfunction duration of 1, we increased it to 2 and thus effectively had a minimum duration of 2 for the malfunctions.

As mentioned in section 2.3 we consider two types of time extensions. The first one is departure time extension, which we model in the following way: when a train is put on the grid at its departure location, we give it a malfunction probability of 0.05. This ensures that on average 5% of the trains experience a delay at their departure location. The expected delays are 0.2 minutes for slow regional trains, 0.5 minutes for fast regional trains and 1 minute for long distance trains. To determine whether a train is a slow regional, fast regional or long distance train we assume that the composition of trains is proportionally the same as in the realistic schedule from section 2.2, i.e. we have 48% slow regional trains, 32% fast regional trains and 20% long distance trains.

The second type of time extensions is running time extension, for which we modify the probability that a malfunction occurs to 0.2 divided by the length of the route, if the agent is a slow regional train, and to 0.5 divided by the length of the route, if the agent is a fast regional or long distance train. By doing so we enforce that on average 20% of slow regional trains and 50% of fast regional and long distance trains experience a delay during their travel. The expected delays are 1.3 minutes for slow regional trains, 2 minutes for fast regional trains and 5 minutes for long distance trains.

The implementation of malfunctions that we used consists of the two classes "ParamMalfunctionGen" and "TestMalfunctionGen", which can be found in the same file[6].

3.6 Randomly generated environments

Alternatively to the fixed railway grid from section 3.1 we also need randomly generated environments to do Machine Learning. The reason is that in a fixed environment the agent could simply learn to exploit specific details of the infrastructure, but our agent should be able to generalise on previously unseen environments. These randomly generated environments should be as realistic as possible. Therefore, we set up some rules for the environment generation.

In Flatland there exists an implementation to generate random environments called "sparse_rail_generator"[7] that serves as a good starting point, but there are two issues that need to be addressed. First of all the randomly generated environments are always loops in the sense that there are no cities that serve as a dead end. Trains can always enter and exit cities at both ends of the city. That should be changed, if the city's two closest neighbour cities are both located in the same direction. In this case trains should have to leave through the exit pointing towards the closest neighbour cities and drive towards one of them.

By doing so we run the risk of creating multiple clusters of cities, which are connected internally, but not from one cluster to the other. To address this problem we integrated the function "contains_cliques"[8] that checks the feasibility of the randomly drawn city positions. If they are located in such a way that they would be split into separate clusters, we deem the positions unfeasible and draw new ones until we have generated city positions that don't split into separate clusters.

The second issue is that the randomly generated rails between cities aren't parallel, but all over the place. If there are two outgoing tracks from one city and two incoming tracks in another city, we would like these two tracks to be connected and to be parallel for the whole way from one city to the other. The first step here was to make sure that the correct tracks are connected. If a train leaves a city on the right hand side track and follows this track to the neighbouring city, it should also enter the neighbouring city on the right hand side and not the left one.

The second step was to choose the order, in which rails are placed from one city

[6] https://gitlab.com/lcleicht/flatland/-/blob/master/flatland/envs/malfunction_generators.py
[7] https://gitlab.aicrowd.com/flatland/flatland/-/blob/master/flatland/envs/rail_generators.py
[8] https://gitlab.com/lcleicht/flatland/-/blob/master/flatland/envs/rail_generators.py

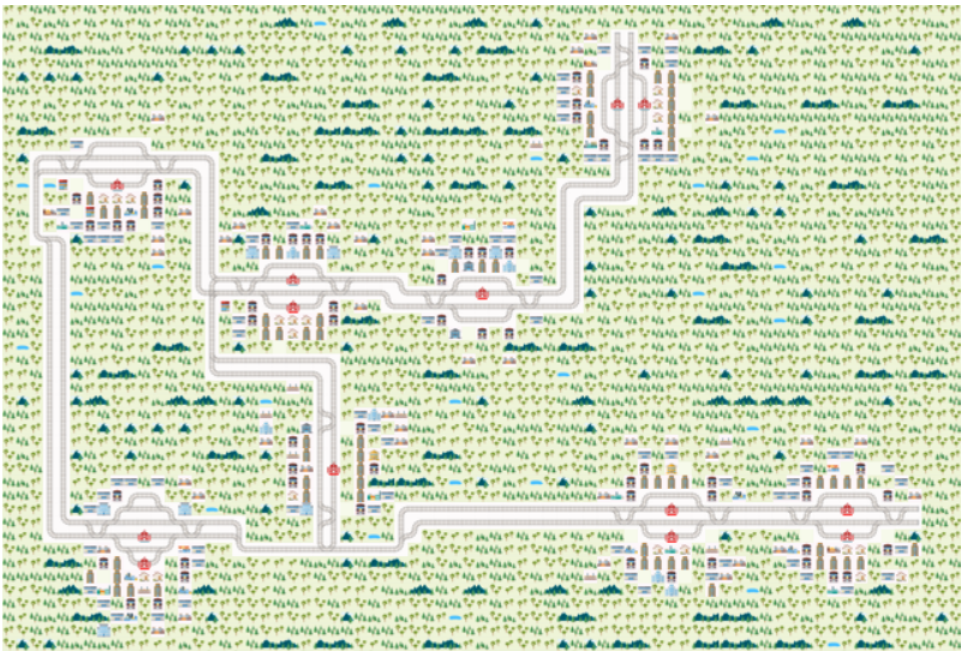
to its neighbour. Since the rail cells between two cities are chosen using a variant of the A* search algorithm, the order of the rails is of utmost importance, because the first rail between two cities can choose the best line according to the algorithm, but the second rail has to take the already existing first rail into consideration and try to avoid it, if possible. We found a rule for the order of the rails that works well to build parallel tracks except in the case, when there are two consecutive turns in the same direction. In that case the two tracks get squashed into a single track, see for example the top left corner of subfigure (a) in figure 3.3, but this is due to the search algorithm and can't be solved by adjusting the order of the tracks.

Placing the tracks in the correct order is necessary to enable right hand side traffic, but not sufficient, because the departure and arrival platforms in the cities are still random and can lie on the left hand side. In this case the shortest path for a train could easily lead it onto a track with a lot of oncoming traffic, especially when the grid layout is a loop and most of the shortest paths use the inside track. Therefore, to give an incentive for traffic to use the right hand side tracks, we always divided the platforms in each city into two groups and chose a departure or arrival platform from the group that would lie on the right hand side with respect to the entry or exit point on the train's path. This procedure seemed to increase the percentage of right hand side traffic, but didn't guarantee it due to the randomness of the environments. For such a guarantee one could determine driving directions during the generation of the environment, when the cities are connected. But since during conflict resolution the trains should be able to reroute on the left hand side track to avoid possible conflicts, we decided not to enforce driving directions on any rail.

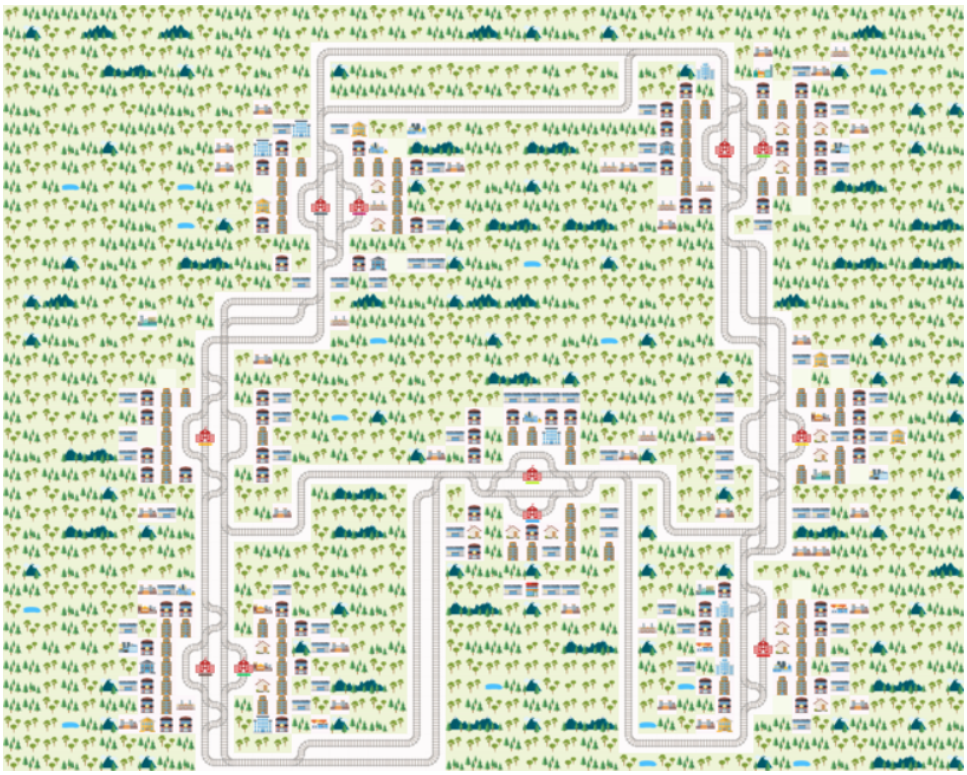
In the figure below we compare two randomly generated environments with width and height sampled uniformly as an integer between 35 and 100. The first one was generated after implementing these new rules. It can be seen that right hand side city exits are always connected to right hand side city entries and there are no unnecessary rail crossings like in the second environment. Also the total number of rails is lower in the first environment, because there are always at most two parallel tracks, while in the second environment there are at some points three parallel tracks. Additionally, we can see two cities that serve as dead ends in the first figure, while there can't be any cities like that with the original method.

The implementation to generate these random environments consists of the "generate" function of the class "SparseRailGen"[9] and all its helper functions.

[9] https://gitlab.com/lcleicht/flatland/-/blob/master/flatland/envs/rail_generators.py



(a) A random environment generated following our rules



(b) A random environment generated with the basic method

Figure 3.3: An example of two randomly generated environments, the upper one has our rules implemented while the lower one doesn't

Chapter 4

Conflict detection and resolution

In this chapter we will discuss the different types of conflicts that can arise during railway operation and methods how to detect and resolve them. Additionally we will look into the computational and conceptual reasons that make conflict detection and conflict resolution challenging.

4.1 Conflict detection

In railway operation we can have two basic types of conflicts between two trains depending on their direction of travel. First a train might block the other's movement. This can happen for example when both trains want to use the same part of the infrastructure at the same time e.g. a junction or a switch, but they aren't blocking each other's exit, or when two trains are following each other and the first one is either slower or experiencing a malfunction. A Flatland example of two trains following each other and the first one experiencing a malfunction is shown in figure 4.1.

Second a deadlock may occur, when both trains are blocking each other, because they both want to use the part of the infrastructure, where the other train is currently located, so they restrict each other's movement to a full stop. An example how this might look in Flatland is shown in figure 4.2.

The difference between the two types of conflicts is that the first one can be resolved by waiting until the first train has overcome its malfunction or made some progress at its lower speed so that the second train can also continue its travel while a deadlock can't be resolved in place and needs to be prevented beforehand for example by rerouting one of the trains so that both use the right hand side track.

In the following we assume that each train has a planned trajectory determined

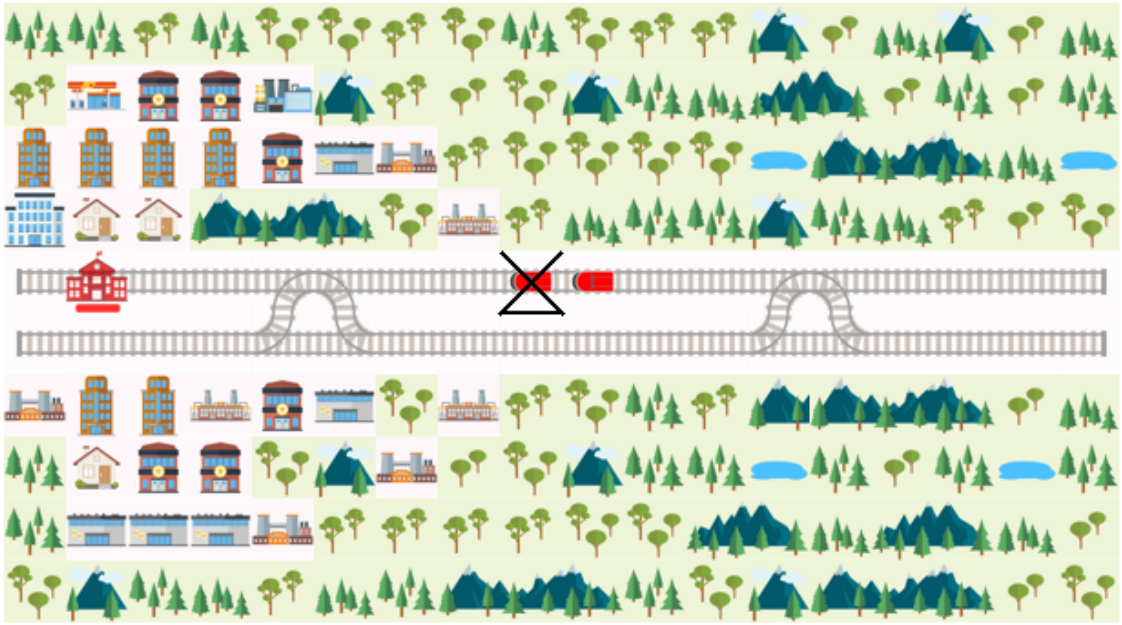


Figure 4.1: An example of a blocking due to malfunction

by the schedule, which it adheres to unless the trajectory is changed due to the resolution of a conflict. We will only consider conflicts between two trains, because conflicts that involve more than two trains can be broken down into several conflicts between two trains. Now if we are given the trajectories of two trains, we can check for conflicts simply by comparing the parts of the infrastructure each train wants to use at every time step and see if there is an overlap. If we find an overlap, we can distinguish between a blocking and a deadlock by checking whether both trajectories contain the other's current position as the next incremental step or not. By checking for overlaps we can also make sure that trains keep a minimum headway between them, if we choose the overlap big enough.

In our implementation we used an overlap of size 2, which means that in order to be conflict-free the trains have to keep two empty cells in between them. It consists of the function "find_conflicts" of the class "PlanningController"[1] and the helper function "find_conflicting_trains" in the same file.

4.1.1 Challenges for Conflict Detection

The biggest challenge for conflict detection is scalability. If a train has a malfunction, we compare its new trajectory to the trajectories of all other trains to check for

[1] https://gitlab.com/lcleicht/flatland/-/blob/master/flatland/envs/planning_controllers.py

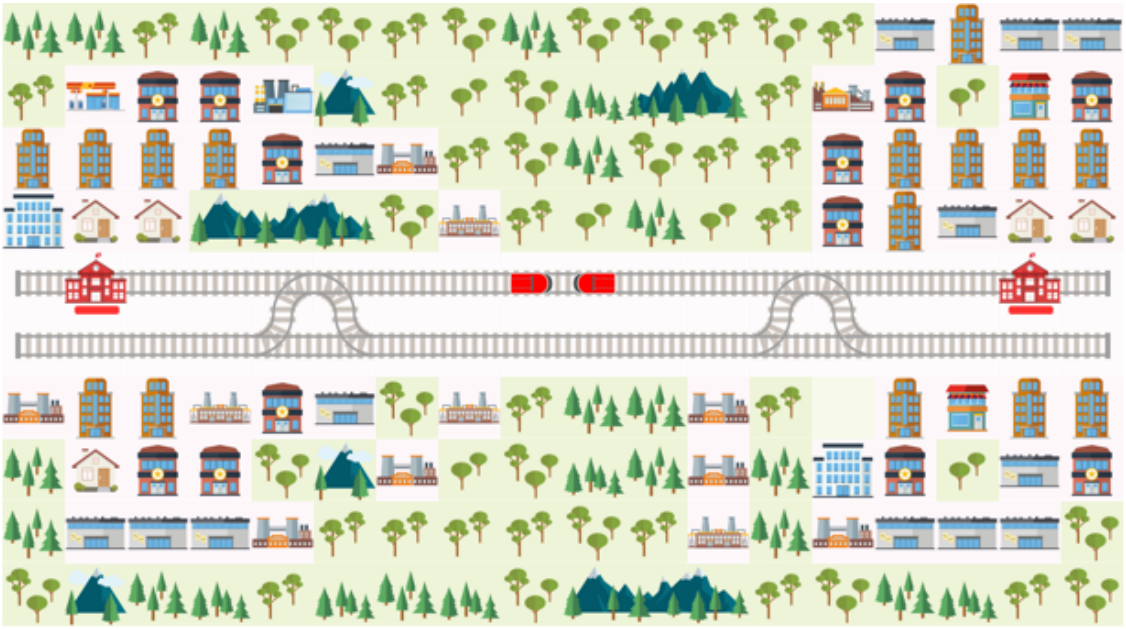


Figure 4.2: An example of a deadlock

conflicts, so conflict detection lies at least in $\Omega(n)$, where n is the number of trains. Additionally, the computation becomes more complex in larger environments, because the trains' routes become longer, so conflict detection has to be done for more time steps. To remedy this bad scaling effect we only do conflict detection for an appropriate time window. Since a train can't have a conflict as long as it isn't put on the grid, we don't need to check for conflicts prior to the train's earliest departure time. Similarly once a train has reached its target and is taken off the grid, we can stop the conflict detection, because the train will remain off the grid for the rest of the episode.

Additionally, we keep track of the conflicts using a list ordered by the time of detected conflicts. As long as the list is empty and all trains are following their planned trajectories there is no need to do conflict detection. Only when a train's trajectory changes due to a malfunction or as the result of a conflict resolution, we need to run the conflict detection and insert all conflicts into the list. When a conflict resolution is chosen, its follow-up conflicts are automatically inserted into the list to be resolved in the next time step. Resolved conflicts are removed from the list. For prolonged periods of smooth railway operation without malfunctions this means that we don't have to do any conflict detection at all.

4.2 Conflict resolution

According to Qu et al. [QCL15](page 660) there are four kinds of conflict resolution:

- Retiming,
- Rerouting,
- Reordering and
- Cancelling connections.

Retiming means to shift the trains' space-time trajectories in the time dimension, rerouting means to shift the trajectory for at least one of the two trains in the spatial dimension, reordering means to change the order, in which the trains are supposed to access the same part of the infrastructure and cancelling connections means that trains scheduled to depart after the arrival of one train at its target station won't wait for the arrival of the delayed train. Again since in Flatland we don't have connections, we will only consider the first three resolution methods and since reordering essentially is either retiming, rerouting or a combination of both, we focus on the first two methods.

Let us illustrate the first two methods with a simple example. In figure 4.3 there is an imminent conflict due to the first train experiencing a malfunction and thus blocking the shortest path of the second train to its target. The second train now has two options to resolve the conflict namely retiming or rerouting. Retiming can in principle be done at any step between the second and first train, but we will focus on the two extreme cases. Either the second train waits at its current position until the malfunction of the first train has been resolved and they both can continue driving while keeping the minimum headway between them or the second train drives straight ahead as far as possible and waits at a distance of the minimum headway behind the first train until it clears the blocked infrastructure. The latter option is usually preferable, because then the trains are bundled up together and leave more space for other trains to maneuver. Rerouting is also possible, because the second train can still use the switch in front to go around the malfunctioning train via the left hand side track.

In this easy example we can quickly find the optimal solution to resolve the conflict. Since the route via the other track is two cells longer, the second train should take it, if it has to wait three or more time steps behind the first train. If it has to wait only one time step, it should drive straight ahead or wait at the current

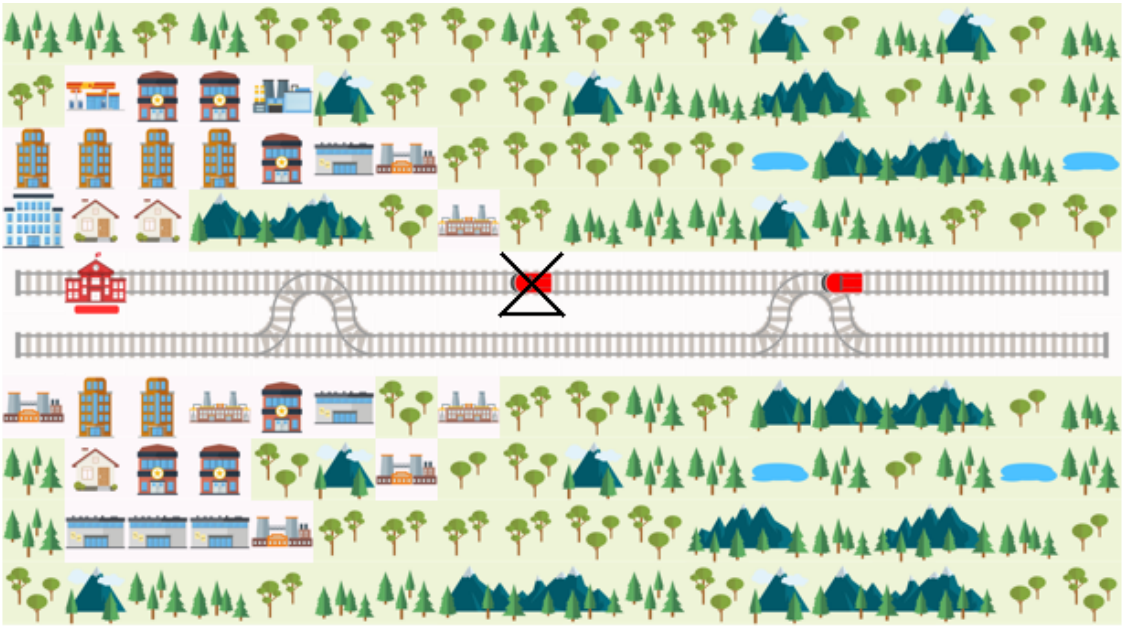


Figure 4.3: An imminent conflict that can be resolved by retiming or rerouting

position. If it has to wait two time steps, the decision will have no impact on the arrival time, but in this case we will always try to stick to the original trajectory as much as possible, so the second train should drive straight ahead and wait two steps behind the first train.

Of course this easy example can very quickly become much more complicated with the addition of more trains. For example there could be oncoming traffic on the other track, which the second train would force to stop by choosing to reroute.

4.2.1 Challenges for Conflict Resolution

While the conflict in figure 4.3 can easily be resolved for example by making the second train wait, we have to make sure that the trajectory can't be changed afterwards in a way that destroys the chosen conflict resolution. We will give an example to illustrate this problem. Imagine that there is a third train driving at a distance of the minimum headway behind the second train. If we now make the second train wait due to a conflict with the first train, this will cause a conflict with the third train, because the third train can't keep driving without violating the minimum headway. An easy solution for the conflict between the second and third train would be for the second train to not wait and simply keep driving. However, then the second train is trapped in an endless cycle of conflicts with the first and

third train respectively that can never be resolved. So to address this problem we made sure that trajectories can't be adjusted until the end of the waiting time, if a retiming solution is chosen, and also not until the end of the malfunction, if they are experiencing one.

A second challenge is to define the number of conflict resolutions that we are computing. We already said that retiming could be done at any step between the time of conflict detection and the time of the conflict, but we will restrict those to the two extreme cases. Similarly for rerouting we could check the shortest n paths for both trains involved. In the easy example with only two tracks checking for more than two paths isn't sensible, since there are at most two different paths to the agent's target, if we exclude reversing at the dead ends. However, in more complex environments trains can have many different routes to their targets.

In order to find a good balance we restrict ourselves to ten resolution options, of which four are retiming options (at the current position or as close as possible to the conflict for both trains) and six are rerouting options (the three shortest paths for both trains involved in the conflict). Note that this doesn't mean that there are always ten possible solutions. In the example of figure 4.3 the conflict can't be solved by retiming or rerouting the first train.

Ideally we would also have a combination of retiming and rerouting, for example wait a few steps until oncoming traffic has cleared the other track and then reroute, but since these combinations blow the space of resolution options out of proportion, we won't consider them yet and leave them for future work.

The implementation of the double track consists of multiple parts. First the rail grid is defined in "make_double_track"[2], which is then used in the function "generate" of the class "TestRailGen"[3]. The conflict detection and the conflict resolution are done in "find_conflicts" and "find_solutions" of the class "PlanningController"[4] respectively. The latter function gathers waiting resolutions from "solution_by_waiting" and rerouting resolutions from "solutions_by_rerouting", all in the same file.

[2]https://gitlab.com/lcleicht/flatland/-/blob/master/flatland/utils/simple_rail.py
[3]https://gitlab.com/lcleicht/flatland/-/blob/master/flatland/envs/rail_generators.py
[4]https://gitlab.com/lcleicht/flatland/-/blob/master/flatland/envs/planning_controllers.py

Chapter 5

Reinforcement Learning

This chapter will serve as a brief introduction to Reinforcement Learning. We will discuss its general idea and some of its methods and algorithms that we can use in the context of railway operations. This chapter is mostly based on the introductory book by Sutton and Barto [SB18].

5.1 General idea

The topic of Machine Learning is often divided into three parts: supervised learning, semisupervised learning and unsupervised learning. In supervised learning the data is labelled, which could be for example a correct class in a classification problem or a correct output value in a regression problem. In semisupervised learning only some of the data is labelled and in unsupervised learning none of the data is labelled. This means that in the latter case a learning agent can't rely on an expert telling them how good or bad a chosen action was. Instead the agent needs to also learn to judge its own actions.

At first glance it might seem like Reinforcement Learning and unsupervised learning are the same thing, but there is a distinction between them. As Sutton and Barto point out ([SB18], page 2) unsupervised learning is often about finding hidden structure in the data, while the goal of Reinforcement Learning is to maximise a reward function.

The idea of Reinforcement Learning is to have a training period, during which the agent can learn by trial and error using the reward function as feedback. Note that in the case of sparse rewards the rewards aren't known for every incremental time step, but are only awarded after a sequence of actions, sometimes even only after a whole episode.

5.2 Markov Decision Processes

The problem of railway operation is a full Reinforcement Learning problem in the sense that actions affect both immediate rewards and also future situations. Its discrete Flatland version can be modelled as a Markov Decision Process (MDP). A MDP consists of an agent and an environment. The agent can interact with the environment by choosing to apply an action from an action space. The chosen action will give rise to a reward, which is passed as feedback to the agent, and may change the state of the environment, which is also observable by the agent. An example for a changing state in railway operation is a simple rail grid with just one train on it. If the train moves forward, its position on the grid will change and therefore also the state of the problem. The general layout of a MDP can be seen in the figure below.

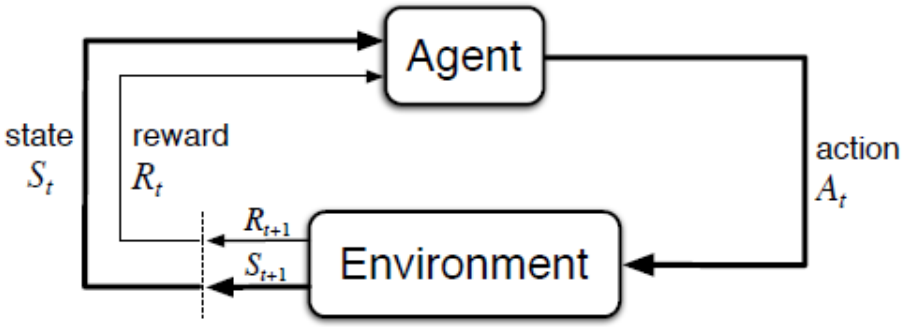


Figure 5.1: The general layout of a MDP from [SB18] (page 48)

The MDP is called a finite MDP, if the set of possible states $\mathcal{S}$, the set of possible actions $\mathcal{A}$ and the set of possible rewards $\mathcal{R}$ are all finite. In this case we can define transition probabilities

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

that fully characterise the MDP in the sense that the probabilities of all possible values of S_{t+1} and R_{t+1} can be computed given the current state S_t and the chosen action A_t. This also means that the state fulfills the Markov property

$$Pr\{S_{t+1}|S_t\} = Pr\{S_{t+1}|S_t, S_{t-1}, \ldots, S_0\}.$$

5.3 Rewards and returns

One way to decide which action to select given the current state is to use q-values. Q-values are approximations to the expected reward of an action

$$q(s, a) \approx \mathbb{E}[R_{t+1}|S_t = s, A_t = a]. \tag{5.1}$$

In other words, if at time step t the MDP is in state s and action a is selected, then the right hand side of (5.1) is the true expected reward at time $t + 1$. There are two problems with this equation. First the true expected reward isn't known, so there is always an approximation error, and second the goal of Reinforcement Learning isn't to choose the action with the highest expected reward at every time step, which could get stuck in a local maximum, but instead to choose the action that maximises the expected return $\sum_{k=1}^{T} R_{t+k}$, so the sum over all rewards in the future, where T is the time of termination of the episode. The expected return can be infinite, if for example there is no time of termination and the rewards are constant and nonzero for every time step. Since in this case we would be stuck with infinite computation time, we apply the principle of discounting.

Discounting means that we don't completely ignore future rewards, but also don't give future rewards the same importance as immediate rewards. Instead for every time step in the future we discount the reward by multiplying with a discount factor $0 \leq \gamma \leq 1$. The goal is then to maximise the discounted return

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

The discount factor guarantees that the sum converges for bounded rewards even when the episode has no termination time. Therefore our goal is no longer to approximate the expected reward, but instead the expected discounted return of an action given a state, which is denoted by

$$q^*(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+1+k}|S_t = s, A_t = a\right]. \tag{5.2}$$

5.4 Policies

One method to approximate the expected discounted return from the previous section is to use policies. A policy maps a tuple consisting of a state and an action to a probability that this action is taken given the state. The corresponding equation is

$$\pi(a|s) = Pr_\pi(A_t = a|S_t = s).$$

Note the index π, because in general different policies assign different action probabilities even when the original state is the same. We can now define an approximation to equation (5.2) as the expected discounted return when following policy π, which is also called the action-value function for policy π

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+1+k}|S_t = s, A_t = a\right]. \tag{5.3}$$

Similar to the action-value function we can also define the state-value function for policy π as

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+1+k}|S_t = s\right]. \tag{5.4}$$

The state-value function $v_\pi(s)$ serves as an indicator of how good a state s is when following a policy π. We now assume that we are in the special case of deterministic states, i.e. when we know the current state s and the chosen action a, we also know the next state s' and the corresponding reward r. Under this assumption we can rewrite equation (5.4) to

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\left[R_{t+1} + \sum_{k=0}^{\infty}\gamma^{k+1}R_{t+2+k}|S_t = s\right] \\
&= \sum_a \pi(a|s)\cdot\left(r(s'(a)) + \mathbb{E}_\pi\left[\gamma\sum_{k=0}^{\infty}\gamma^k R_{t+2+k}|S_{t+1} = s'(a)\right]\right) \\
&= \sum_a \pi(a|s)\cdot(r(s') + \gamma\mathbb{E}_\pi\left[G_{t+1}|S_{t+1} = s'\right]) \\
&= \sum_a \pi(a|s)\cdot(r + \gamma v_\pi(s')).
\end{aligned}$$

The last line is a special case of the Bellman equation, if the probability that choosing action a in state s leads to the next state s' with reward r is equal to 1. We still need to define the optimal action-value function

$$Q^*(s,a) = \max_\pi \mathbb{E}[G_t|S_t = s, A_t = a, \pi], \tag{5.5}$$

which also obeys the Bellman equation in the sense that if the optimal action-values for all possible successor states s' with possible actions a' were known, the optimal action-value for the current state can be computed as

$$Q^*(s,a) = r + \gamma\max Q^*(s',a'), \tag{5.6}$$

where r is the reward generated by selecting action a in state s.

5.5 Exploration versus exploitation

One of the most important aspects of Reinforcement Learning is the trade off between exploration and exploitation. At every step of a MDP the agent can choose either an exploring or an exploiting action. An exploring action is usually a random action drawn from all possible actions with a uniform distribution, but there are also different exploring approaches such as always choosing the action that has been selected the least amount of times in the past. An exploiting action is the action that currently has the highest action value, so it is selected as $\arg\max_a q_\pi(s, a)$.

The most common approaches to decide between exploration and exploitation are greedy, ϵ-greedy and simulated annealing. Greedy action selection always chooses exploitation and never exploration and ϵ-greedy action selection chooses exploration with a probability of $0 < \epsilon < 1$ and exploitation with a probability of $1 - \epsilon$. Greedy actions maximise the reward over one step, but exploration may lead to more refined action values and thus greater rewards in the long run, which the ϵ-greedy actions try to achieve. However, ϵ-greedy action selection never stops exploring, even when the action values aren't changing anymore due to the fixed percentage of exploring actions.

Finally, simulated annealing also uses an ϵ as the probability to select an exploring action, but the value of ϵ isn't constant and can change over time. It starts with a large value, sometimes even 1, and then decays over time for example exponentially. The simulated annealing approach resembles intuitive human learning, because without any information we learn by trial and error until we have developed a feeling for good and bad actions, at which point we stop exploring and apply our gathered experience in practice.

5.6 Stochastic Gradient Descent

Before we can apply value functions and action selection in practice to a neural network and thus actually do Reinforcement Learning, we still need to establish how these networks learn. For the computation of an output signal a neural network in every layer applies weights and biases to the input signal and passes this linear combination on to a nonlinear activation function. In supervised learning one can then compare the computed output to the expected output and define its difference as the loss of the neural network.

Learning means in this context to adjust the parameters of the network, i.e. the weights and biases, to decrease the loss. In gradient learning methods this is

done by computing the gradient of the loss, which consists of the partial derivatives with respect to the individual parameters, and then to update every parameter by subtracting the value of the corresponding partial derivative multiplied by a step size. Since this procedure is computationally expensive, the idea of stochastic gradient descent is to randomly choose only one parameter to update in every iteration and keep all others at the same values. In unsupervised learning one has to define the target value of the expected output manually, since no labelled data is available and we will see an example of how this can be done in the following section.

5.7 Deep Q Networks

Combining the theory of the previous sections we are now ready to apply a first Reinforcement Learning algorithm to the Flatland environment. On the Flatland website there are tutorials[1] on how to use a variant of the Deep Q Network (DQN) method for a single agent and for multiple agents. The necessary python files are all contained in the Flatland starter kit[2].

The DQN method was introduced by Mnih et al. [MKS+13] as a Deep Learning model to learn control policies for several Atari games. A DQN is a neural network function approximator with weights θ to the optimal action-value function defined in equation (5.5) and therefore referred to as $Q(s, a; \theta)$. The loss function for the training changes in every iteration and is the squared difference between the computed action-value using the current parameters $Q(s, a; \theta_i)$ and the target, which is the approximation to the right-hand side of equation (5.6) with the parameters of the previous iteration $r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})$. Updates of the parameters are done using stochastic gradient descent from section 5.6 on the loss function. The DQN method also uses a replay buffer with a fixed capacity, in which the agent's experiences are stored as a tuple containing the state, the selected action, the generated reward and the subsequent state. The learning can then be applied to samples drawn from the replay buffer, rather than only from subsequent states in an episode, which tend to be highly correlated.

Over the years several improvements have been made to the DQN method. The first one we will talk about is Double Q-learning. Since using the same Q-function for both evaluation and action selection leads to overestimations of the

[1] https://flatland.aicrowd.com/tutorials/rl.html (accessed on 10.12.2021)

[2] https://gitlab.aicrowd.com/flatland/flatland-starter-kit (accessed on 10.12.2021)

action values, when they can't be accurately approximated for example due to noise, van Hasselt et al. proposed Double Q-learning [VHGS16] with two separate action-value functions, one for evaluation and the other one for action selection. In fact the two functions can still use the same neural network architecture, they only have different parameters. For the evaluation the neural network uses the parameters θ^-, which are updated every few steps as copies of the current parameters θ, but are otherwise fixed.

The second improvement we will talk about is Dueling Network Architectures [WSH+16]. This architecture increases the performance compared to the previous two methods and is also the one used in the Flatland tutorials. In it the computations of the state-value function and an advantage function are separated into two streams. The purpose of the advantage function is to assess the impact of the possible actions. Only afterwards are they combined into a computation of the state-action value.

The benefit of this method is that the network can learn, which states are better than others without learning how actions affect this state. This makes sense for our Flatland railway operation, because most of the time trains are on straight rails and their only two options are to move forward or to stop, both of which have very little impact on the state. However, at a junction actions are very important and can be the deciding factor between a train reaching its target or ending up in a deadlock. By decoupling the state value estimation from possible actions it is updated more often and becomes more robust especially in the case that the number of possible actions is large.

Together these two improvements yield the Dueling Double Deep Q Network (DDDQN) method.

5.8 Imitation Learning

The idea of Imitation Learning is that the algorithm tries to mimic an expert either as closely as possible during the whole training process or as an initialisation, from where it tries to improve. We focus here on the expert initialisation, because we have got baseline solutions on our hands that score better rewards than random actions, but aren't always choosing the best action. Therefore, in this context Imitation Learning means that whenever the algorithm chooses to do exploration, we don't select a random action anymore, but instead the expert solution provided by a heuristic baseline.

5.9 Curriculum Learning

In 2009 Bengio et al. presented their paper about Curriculum Learning [BLCW09]. Since humans and animals learn better, when they are presented with tasks of increasing difficulty, they reasoned that the same principle could help to improve the convergence speed and the quality of Machine Learning. A curriculum starts with easy training examples and then gradually increases the difficulty to guide the training process towards better regions in the parameter space. In this context the principle can be applied to the size of the environment and the number of trains per episode. A few trains on small environments are easier to guide without causing deadlocks and more forgiving regarding bad action selection than many trains that could get stuck in a deadlock and contribute a penalty to the final reward.

Chapter 6

Experimental results

In this chapter we discuss the evaluation setup that we used and the different algorithms that we compared. The experiments consist of three parts. The first part is done on the simple double track from chapter 4, the second part is done on the realistic railway grid from section 3.1 and the third part is done on the random environments discussed in section 3.6. In each part we evaluated conflict resolution performed by an RL agent in comparison to conflict resolution performed by two heuristic baseline solutions.

6.1 Hyperparameters

In addition to all the usual hyperparameters of Machine Learning, we also have to adjust the parameters for the environments. These are the width and height of the environment, the number of trains, the number of cities, the number of rails in and between cities, the length of the episode, the malfunction rate and the initial schedule. Each of them can have a big impact on the outcome of the learning process. For example when we had a high malfunction rate on the simple double track grid, the algorithm learned quickly to never reroute and only wait, possibly because the risk of having a malfunction while rerouting via the other track and thus blocking all oncoming traffic was too big.

Some of the environment parameters had to be adjusted to the task at hand. For instance the initial schedule for the double track and the random environments was the conflict-free schedule with a stochastic buffer time as suggested in section 3.4. However, the distribution of the buffer time and also of the malfunctions gravely affected the possible choices for conflict resolution. While a truncated normal distribution with a mean of five time steps gave a sufficient buffer time for random

environments, so that usually both retiming and rerouting options were possible, the same distribution led to too tightly packed schedules on the double track, so that rerouting wasn't possible most of the time and had to be changed to a mean of ten time steps.

Similarly, the realistic malfunction probabilities introduced in section 3.5 were sufficient on random environments to cause conflicts with other trains' trajectories. On the double track we tried different values for the expected delay and found two minutes to give a good trade-off between causing conflicts at all but not blocking the track for too long so that only waiting is an option. This means that we assumed there were only fast regional trains on the double track.

In the following table 6.1 we give an overview over the five different training environments and the choice of their hyperparameters. The asterisk in the row describing the length of the episodes means that the length is different for every episode. It is computed as the latest planned arrival time of all trains plus a buffer time equal to five times the number of agents to allow for delays caused by malfunctions. The three rows for the malfunction rate and the expected delay describe in this order slow regional trains, fast regional trains and long distance trains. The notation $T\mathcal{N}(\mu, \sigma^2)$ for the buffer time distribution means that it follows a truncated normal distribution with mean μ and variance σ^2 and can take values between zero and two times the mean rounded to the nearest integer.

Hyperparameters for the different environments					
Hyperparameter	Double Track	Model grid	Random (Small)	Random (Medium)	Random (Big)
Width x Height	18x10	118x27	35x35	50x50	60x60
Max. number of cities	2	3	4	6	8
Number of trains	20	50	20	35	50
Max. number of rails in city	2	11	4	6	6
Length of episode	*	1500	*	*	*
Malfunction rate (during travel)	0.2/15	0.2/path length	0.2/path length	0.2/path length	0.2/path length
		0.5/path length	0.5/path length	0.5/path length	0.5/path length
		0.5/path length	0.5/path length	0.5/path length	0.5/path length
Expected delay (during travel) [min]	2	1.3	1.3	1.3	1.3
		2	2	2	2
		5	5	5	5
Buffer time distribution	$\sim TN(10,\ 10/3)$	-	$\sim TN(5,\ 5/3)$	$\sim TN(5,\ 5/3)$	$\sim TN(5,\ 5/3)$

Table 6.1: The hyperparameters used for training and testing on different environments

6.2 Training

We adapted the DDDQN algorithm introduced in section 5.7 to decide between possible conflict resolutions. Each time a conflict was found during an episode the algorithm was given the full action space containing the ten possible resolutions discussed in section 4.2 together with the information, whether each of the possibilities was feasible in the specific conflict or not. Between those ten resolutions it had to choose one either by exploration or by exploitation, see section 5.5.

The state consisted of the set of possible conflict resolutions, each containing the trajectories of both trains for the next 30 steps, the number of follow-up conflicts caused by the particular resolution as well as the time and type of those follow-up conflicts and the handles of the involved trains. In case that the conflict resolution isn't feasible, the corresponding part of the state is filled with zeroes. We also apply this zero padding to the follow-up conflicts, if there are fewer than the maximum number of follow-up conflicts, which we set to the nearest integer of the number of trains divided by five. The reward associated with a state-action pair is either the reward of the following time step or the end reward of the episode, if the conflict is the last one for that episode.

We trained an instance of the DDDQN algorithm on the double track and each size of the random environments from table 6.1, except for the big random environment due to an inexplicable error. We also tried the ideas of Imitation Learning and Curriculum Learning, see sections 5.8 and 5.9. For the Imitation Learning instances we select all exploratory actions according to the heuristic baseline solution, which we will discuss in detail in the next section.

Curriculum Learning in this context means that instead of having three separate algorithms for the three different sizes of random environments, we only take one algorithm and let it start on small random environments. After a while, when the learning process shows no more improvement, we increase the complexity of the environment. Since the network encounters a greater variety of environments during training, it is supposed to generalise better than the three separate instances.

In order to prevent overfitting, we implemented a termination condition to stop the training, if there was no more improvement for 30 consecutive episodes in the smoothed completion rate. We chose the completion rate over the total reward, because the reward is heavily influenced by outliers, especially episodes, in which not all trains reach their target. The penalty for a few trains not reaching their target makes a big difference in the reward of the episode even if the completion rate is still close to 100% and even after smoothing this difference usually can't be

recovered within 30 episodes.

The code used for the training are the two functions both called "train_agent" for random environments in the file "multi_agent_random_env"[1] and for the double track in "multi_agent_test_env"[2].

6.2.1 The deep network

The DDDQN algorithm trains the weights and biases of a deep network, which by default has two fully connected hidden layers and uses the Rectified Linear Unit (ReLU) activation function. In our training we used a more complex network with five fully connected hidden layers, batch normalisation and dropout of 10% after every hidden layer and the leaky ReLU activation function. We got better results with the more complex network, but in the interest of time we didn't check, how much each change was contributing to increase the quality of railway operation.

The implementation of the deep network is given in the class "DuelingQNetwork"[3].

6.2.2 Visualisation of the training process

In this subsection we show two examples of the training process of our deep networks, one that was trained on small random environments without any additional modifications and one that included both Curriculum Learning and Imitation Learning. The graphs of the other networks look similar only with different axes scalings and can be found in the appendix A.2.

The graphs are all smoothed with a smoothing coefficient of 0.99. Even so the graph showing the reward is very volatile, because of the high variance in the total rewards, which aren't all directly linked to the conflict resolution selection. Simply a different number of malfunctions or a different total length of all trains' paths can cause different total rewards for subsequent episodes. As mentioned in section 6.2 this is the reason, why we chose as a termination criterion the completion rate, which initially increases during training, but eventually levels off. The jumps in the graphs with Curriculum Learning signify that the termination criterion has been

[1] https://gitlab.com/lcleicht/flatland/-/blob/master/flatland_starter_kit_master/baselines/reinforcement_learning/multi_agent_random_env.py

[2] https://gitlab.com/lcleicht/flatland/-/blob/master/flatland_starter_kit_master/baselines/reinforcement_learning/multi_agent_test_env.py

[3] https://gitlab.com/lcleicht/flatland/-/blob/master/flatland_starter_kit_master/baselines/reinforcement_learning/model.py

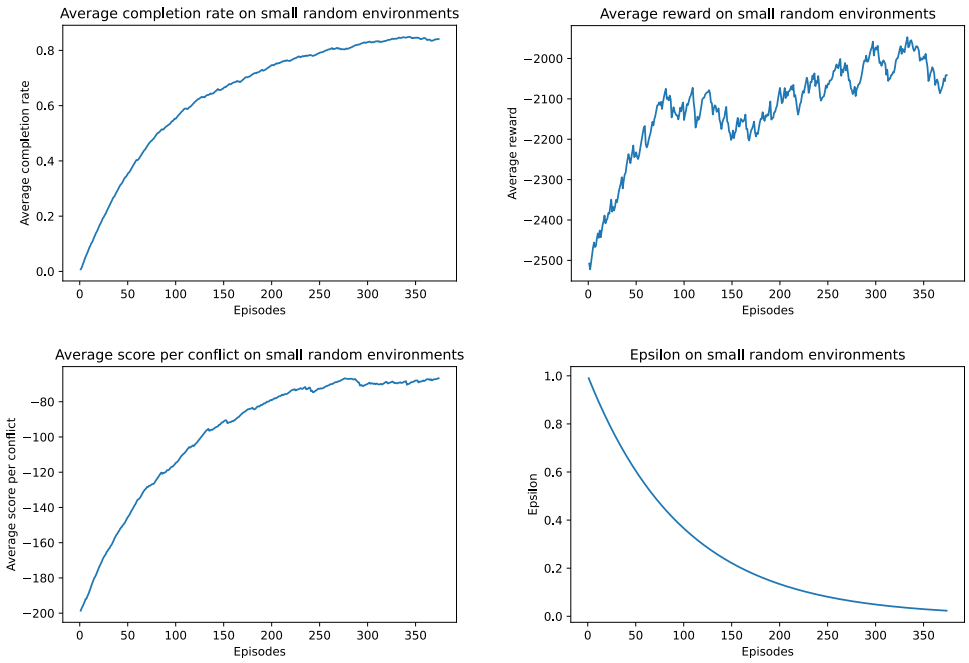


Figure 6.1: Visualisation of the training process

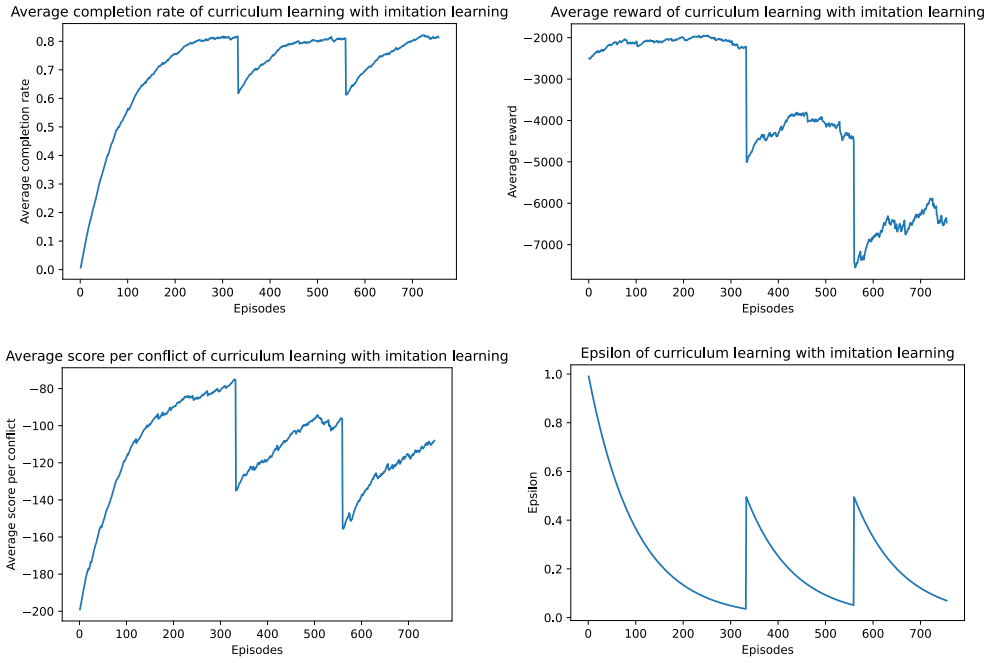


Figure 6.2: Visualisation of the training process with Curriculum and Imitation Learning

fulfilled for the current size of the environments and the training then switches to larger environments thus reducing the completion rate, the score per conflict and the total reward, because there are more trains with longer paths, so more malfunctions occur. At every change in environment size we reset the epsilon value to 0.5 to give the algorithm the possibility to explore action selection on the larger environments. Otherwise the epsilon value decays exponentially.

6.3 Baseline solutions

We compare our trained networks with three baseline solutions. The first is very simple, it always chooses to resolve a conflict by making the train with the higher handle wait in place. The reason for making the train with the higher handle wait is that it has a later departure time, so it should wait until the trains that were previously put on the grid have released the blocked infrastructure. Since no rerouting is ever done and the initial schedule is conflict-free, the choice of making a train wait only shifts the trajectories in the time dimension. As long as the waiting resolution is valid there won't be any deadlocks, only the episodes might take unnecessarily long.

The second baseline solution is a little bit more sophisticated. It uses a heuristic to give the different possible resolution options a value that acts as a penalty term, because it then chooses the resolution with the smallest value. To compute the penalty we consider the conflict at hand as well as the first order follow-up conflicts. If a resolution has no follow-up conflicts, then its value is simply the time spent waiting for a waiting resolution or the difference in the length of the routes for a rerouting resolution. However, if the resolution causes some follow-up conflict, we check whether that follow-up conflict is solvable by waiting and give it a corresponding penalty. If the follow-up conflict is a deadlock, it can't be resolved by waiting, so it receives an infinite penalty, otherwise it receives as a penalty once again the waiting time.

The third baseline solution is the worst case scenario, in which no conflict resolution is chosen whatsoever, which is why we refer to it as the do nothing baseline. In this case the trains will crash into each other, if they have conflicting trajectories.

The implementation of the action selection by the first two baseline solutions are the functions "always_choose_wait" and "choose_solution_manually" both in the class "PlanningController"[4].

[4]https://gitlab.com/lcleicht/flatland/-/blob/master/flatland/envs/planning_controllers.py

6.4 Evaluation

For the evaluation we chose four indicators to compare the quality of railway operation. The first one is obviously the total reward, because it estimates the passengers' dissatisfaction. The second one is the completion rate, since we would like all trains to reach their destination eventually. The third one is the reward per conflict, because the conflict resolution should be chosen in a way that adds for each conflict as little negative reward as possible. And finally as a fourth indicator we count the total number of conflicts, since we would like conflicts to be contained and resolved locally and not spread through the whole grid through follow-up conflicts.

All these indicators are average values over 100 episodes in the case of the double track, small and medium random environments and over 50 episodes in the case of big random environments and the custom grid due to the long computation time of the larger environments.

The evaluation of the DDDQN instances is done using masking for the action selection. This means that we eliminate invalid conflict resolutions even if the deep network assigns them the highest value. We order the ten conflict resolution options by the value computed by the deep network and then choose the first valid option or none, if all are invalid.

The results of the evaluation are shown in the following table 6.2. The rows show the average values of the different indicators total reward, completion rate, reward divided by number of conflicts and number of conflicts. The implementation of the evaluation is the function "evaluate_agents"[5].

6.4.1 The three baseline solutions

Comparing the waiting and heuristic baseline solutions the heuristic achieves slightly better rewards than the waiting baseline in most episodes. However, in very complicated episodes it sometimes chooses a conflict resolution that leads to a deadlock and thus a bad reward, even if the deadlock was preventable and the waiting baseline did finish the episode without a deadlock. Therefore, the completion rate of the waiting baseline is slightly higher, even though it creates more follow-up conflicts than the heuristic.

On the custom grid the waiting baseline achieves terrible results. The reason is that the initial timetable on the custom grid isn't conflict-free. In fact it has three

[5] https://gitlab.com/lcleicht/flatland/-/blob/master/flatland_starter_kit_master/baselines/reinforcement_learning/Multi_Agent_Evaluation.py

	Double track	Random environments			Custom grid
		small	medium	big	
Waiting baseline	-183.0	-1812	-5692	-15681	-53773
	1	**0.917**	**0.8921**	0.8308	0.3233
	-71.26	**-49.42**	**-78.82**	**-124.8**	-308.3
	2.80	42.37	82.25	131.5	203.2
Heuristic baseline	**-181.2**	**-1632**	**-5129**	**-11882**	**-18101**
	1	0.896	0.8776	**0.8646**	**0.7624**
	-61.81	-69.97	-102.6	-181.4	-198.5
	3.83	27.86	55.66	65.13	118.7
Worst case	-1167	-5594	-20841	-57240	-61604
	0.8370	0.4575	0.4003	0.3550	0.2336
	-266.4	-341.6	-681.4	-1075	-739.0
	3.29	**18.23**	**32.9**	**55.25**	**85.28**
Double Track DN	-191.6	-1938	-5372	-15642	-35534
	1	0.8640	0.8750	0.8075	0.5253
	-65.77	-51.06	-81.77	-134.3	-55.39
	3.07	53.27	93.99	164.6	713.5
Double Track	-283.8	-2517	-6627	-16476	-41077
Imitation DN	0.9845	0.7945	0.8286	0.8037	0.4780
	-64.16	-61.49	-114.4	-205.3	-75.83
	5.16	57.53	77.97	107.5	718.3
Small Env DN	-186.5	-2625	-6992	-14891	-25988
	0.9995	0.7900	0.8225	0.8224	0.6449
	-70.84	-63.52	-106.7	-187.6	-148.7
	2.94	61.28	112.2	117.0	325.4
Small Env	-220.9	-2540	-7474	-15659	-26779
Imitation DN	0.9940	0.7975	0.8052	0.8155	0.6396
	-66.98	-59.47	-104.6	-175.9	-147.3
	8.20	67.61	128.8	128.5	518.9
Medium Env DN	-323.6	-2549	-7075	-16454	-36713
	0.9805	0.7975	0.8165	0.7931	0.5159
	-63.47	-60.75	-105.2	-168.5	-63.99
	9.47	81.96	99.66	203.0	1045

	Double track	Random environments			Custom grid
		small	medium	big	
Medium Env	-356.3	-2582	-6632	-16049	-41409
Imitation DN	0.9730	0.8010	0.8289	0.7996	0.4543
	-67.35	-60.56	-96.7	-158.9	**-51.58**
	10.16	71.67	124.5	356.4	1723
Big Env	-316.6	-2524	-7611	-21185	-39953
Imitation DN	0.9790	0.7935	0.7974	0.7443	0.4820
	-55.82	-56.44	-113.9	-153.8	-56.66
	12.8	79.72	109.8	255.6	915.5
Curriculum DN	-276.2	-2708	-7039	-15776	-36182
	0.9860	0.7720	0.8165	0.8082	0.5331
	-64.66	-60.63	-123.2	-175.3	-77.00
	5.60	94.46	90.09	142.4	1192
Curriculum	-269.0	-2524	-7484	-15265	-39344
Imitation DN	0.9890	0.8040	0.8014	0.8131	0.4824
	-66.24	-61.29	-103.8	-156.0	-74.88
	5.19	65.12	129.8	198.1	907.7

Table 6.2: The results of the evaluation of twelve different models on five different environments

initial conflicts that have to be resolved before the start of the episode and one of those three can only be resolved by rerouting. Therefore, it isn't surprising to see the bad results of the waiting baseline.

The do nothing baseline achieves the worst results on any environment as expected. The resulting rewards and completion rates merely serve as a comparison to assess how much improvement the other baseline solutions or trained deep networks make by their conflict resolution selection.

Overall the waiting and heuristic baseline solutions performed better than all deep networks. The best average reward was achieved by the heuristic baseline on all five evaluations sets. The highest average completion rate was achieved three times by the waiting baseline and two times by the heuristic baseline. The lowest average number of conflicts was achieved once by the waiting baseline and four times by the do nothing baseline, but the reason is that conflict detection fails, when trains no longer follow their planned trajectory, which in the worst case happens for almost all conflicts. The highest average reward per conflict was achieved three times by the waiting baseline, once by the heuristic baseline and once by the deep network trained on medium sized random environments with Imitation Learning, but the reason for that is simply the high number of conflicts generated by the deep networks, which causes the absolute value of the ratio to become small.

6.4.2 The deep network trained on the double track

The deep network that was trained on the double track without Imitation Learning was actually the only network to reach a 100% completion rate on its training environment. While this is a good result, the more interesting question is how well the learned parameters generalise on previously unseen environments like the randomly generated ones. It turns out that it still performs best out of all deep networks on small and medium environments. Only on larger environments it gets surpassed by other networks.

This good performance is quite encouraging, because it means that the basic principles of conflict resolution can be learned on very simple environments, where computation is inexpensive. Since the infrastructure in a railway grid is repetitive and there are only a few different types of conflicts, it might be possible to design small training environments, where a network could learn to stick to the 100% completion rate and maximise the reward in a way that will eventually outperform the baseline solutions on larger environments.

6.4.3 The deep networks trained on random environments

The deep networks that were trained on different sizes of random environments all performed very similar, usually within a few percent on all performance indicators. Interestingly enough the size of the training network doesn't guarantee a good performance in the evaluation on environments of the same size. For example out of all networks trained on random environments the best performance on small environments is achieved by the network trained on big environments and vice versa.

6.4.4 Curriculum and Imitation Learning

Imitation Learning didn't seem to make much of a difference. On some environments the Imitation Learning instances performed a few percent better and on other environments they performed a few percent worse. Even though the heuristic baseline can be treated as an expert solution, because it performs better than all other solutions, it doesn't seem to help the learning process of a deep network, when chosen instead of random actions in exploratory moves.

Curriculum Learning also didn't boost the performance, which isn't surprising, because of the similar performance of all the networks that were trained on environments of different sizes. This doesn't mean that Curriculum Learning can't help when applied in a different way, but in this context, where the curriculum was only concerned with the size of the environment and the number of trains, there was no improvement. An example of a different way to apply Curriculum Learning is to learn about the different types of conflicts one at a time. Such a curriculum could start with two trains trying to access the same intersection and having to decide which train has to wait. Afterwards it could learn, when a train is about to drive into a blocked section of the infrastructure, whether to stop and wait or to reroute. Finally it could go on to two trains trying to drive on the same single track in different directions and having to decide, which train has to reroute or wait in front of that part of the grid.

Chapter 7

Outlook and Conclusion

In this chapter we discuss the possible future steps to continue the work of this thesis and conclude the thesis with a discussion of the results.

7.1 A more realistic model

As mentioned in chapter 3 we used the existing simulator of the Flatland challenge for training and testing in the interest of time. However, as soon as a more realistic simulator exists, the analysis of this thesis should be repeated on the new simulator to see, whether the results still hold or new insights can be obtained. There is also the possibility to make the Flatland model more realistic, e.g. by enforcing traffic on the right hand side or by clustering parts of the infrastructure into blocks that can only be accessed by one train at a time. This way one definitely won't win the challenge, but it may help to shrink the gap between simulated and real life train movements.

Other ways to make railway operation more realistic is to introduce characteristics like maximum speed and length to the trains. So far the different types of trains only affect the probability and duration of malfunctions, but all trains drive at the same speed. Different speeds would create a new type of conflict, when a faster train gets stuck behind a slower train, which is similar to the example in figure 4.1, if the first train experiences a malfunction for only a short time. Different lengths would make conflict resolution more complicated, because longer trains need more space to maneuver. The speed of a train is already a train property in Flatland, but different lengths would need to be newly implemented.

An issue with the environment generation is that cities can be far apart and only connected by two tracks, each one dedicated to one driving direction. If the distance

between two cities is too far, we would like to insert possibilities for overtaking so that faster trains don't get stuck behind slower trains for the whole distance. The inclusion of different speeds of trains would have an impact on the conflict resolution selection. Heuristically, the train with the higher speed would have priority over slower trains and should always be favoured in the conflict resolution. Deep networks would need to learn this new type of conflict, when a fast train gets stuck behind a slow train, and how to prevent it from happening.

Combining the issues of environment generation and action controllers future work could also be done to generate environments, especially cities and their switching areas, in a way that allows to use realistic controllers like the Right Hand Side and Cluster Controller from section 3.3 in train traffic simulations. Doing so would enable safe railway operation, during which the randomly generated infrastructure is divided into separate blocks that can only be accessed by one train at a time.

7.2 Initial schedules for random environments

The initial schedules for random environments proposed in section 3.4 serve the purpose of being initially conflict-free. However, that isn't the only criterion for schedules in the real world. Instead schedules are chosen as the solution of an optimisation problem, see for example the book by Hansen and Pachl [HP14] (pages 164-165), where the objective is to select paths for the trains in a way that maximises the financial profit. The constraints ensure on one hand that at most one path is chosen for each train. If no path is chosen for a train, this train is regarded as cancelled. On the other hand the constraints also ensure that no conflicting pair of paths is chosen for any two trains.

Since financial profit isn't yet implemented in Flatland, in our case we would need to modify the objective function for example to minimise the sum of the departure times, so that all trains leave as early as possible. As available paths we could either take the different paths from the trains' origins to their destination or stick with the shortest paths and only change the departure time. Choosing two departure times for trains that would cause a conflict between them or violate the safety distance would be forbidden by the set of conflicting pairs of paths K. The departure times of all trains would need to be between 1 and the maximum number of time steps in the episode t_{max}. If we schedule all trains without cancellation and denote by $P_{t,y}$ the shortest path of train t starting at time step y, a possible

optimisation problem would be

$$
\begin{aligned}
\min \quad & \sum_{y=1}^{t_{max}} y \cdot \sum_{t} X_{P_{t,y}} & \\
s.t. \quad & \sum_{y} X_{P_{t,y}} = 1 & \forall t, \\
& X_{P_{t,y}} + X_{P_{t',y'}} \leq 1 & \forall (P_{t,y},\ P_{t',y'}) \in K, \\
& X_{P_{t,y}} \in \{0,\ 1\} & \forall t, y, \\
& y \in \mathbb{N} \cap [1,\ t_{max}]. &
\end{aligned}
$$

A valid solution for this optimisation problem has to choose for every train a departure time as a positive integer smaller or equal to t_{max}, so that the paths of all trains are conflict-free. An optimal solution is a valid solution that minimises the sum of the departure times.

By keeping the binary variables $X_{P_{t,y}}$, which are equal to 1, if and only if train t is scheduled to depart at time y, the problem is a Set Packing model like the problem defined by Pachl and Hansen and can be solved by the Branch-and-Cut-and-Price algorithm proposed by Cacchiani et al. [CCT08] (pages 10-11).

Since the number of decision variables is the product of the number of trains and the maximum number of time steps in an episode, it doesn't scale well for larger episodes. If the allowed time to find a schedule is restricted, one could alternatively approximate the optimal solution by cooperative pathfinding [Sil05].

On page 118 of this paper Silver points out the sensitivity towards the ordering of the agents. In our approach from section 3.4 we simply took the order, in which the agents were generated, and planned their paths accordingly. More advanced techniques either check multiple orderings like Li et al. did in their winning Flatland solution [LCZ+21] (page 479), vary the ordering dynamically [Sil05] (page 119) or use prioritized planning [Lat91] as suggested by Silver.

7.3 More conflict resolution options

As discussed in section 4.2.1 it would be beneficial and also more realistic to combine retiming and rerouting options into more complex conflict resolution options. It is a heavy restriction that rerouting has to happen immediately after the conflict has been detected or not at all. In the future one could keep track of all trains, which are currently affected by a conflict and check at every time step, whether better resolution options than the currently chosen one have become available.

7.4 Scalability

Despite our best efforts to make the computation of conflict detection and resolution as efficient as possible, see subsection 4.1.1, they still don't scale well with an increasing number of trains and larger grids. If the implemented way of conflict detection and resolution were to be applied to large grids say a country's full network, there would be dire need to reduce the computational effort. One example of a possible improvement that is applicable to larger grids is to skip conflict detection for train pairs that can't have conflicts due to their spatial separation. There is no need to do conflict detection for two S-trains in different cities, if they are both restricted to their own city.

7.5 Zero padding

The way that the state is padded with zeroes might make it unnecessarily hard to learn a good mapping from the current state to the desired action, because there is no difference between a valid action that has no follow-up conflicts and an invalid action. For both the part of the state that contains information about possible follow-up conflicts will be padded with zeroes. Implementing a clear distinction between an invalid action and a valid action with no follow-up conflicts could improve the results of the Machine Learning methods a lot, especially since the number of training episodes until the termination criterion stops the training is usually small.

7.6 Difficulties for Reinforcement Learning

The problem of conflict resolution is in a way harder to tackle by Reinforcement Learning than the Multi-agent Pathfinding problem, because the connection between states and actions isn't as clear. The classical MAPF problem needs the Reinforcement Learning algorithm to act at every time step and choose for each agent whether to move or not. The next state will then be almost the same as the previous state with the only difference that the agents with movement actions will have moved to a neighbouring position.

This is quiet different from the states during conflict resolution, because in our case the algorithm only becomes active, if a conflict has been detected and applies conflict resolution to them in a chronological order. That is to say that even after a conflict resolution action, which induces a follow-up conflict, this follow-up conflict

might not be the one to determine the next state, if a different conflict happens at an earlier time.

Additionally there is the element of stochasticity, since the outcome of a resolution action can change at every time step due to a random malfunction. For example in figure 4.3 the information about the rerouting option is that there won't be any follow-up conflicts. If the train is then rerouted and experiences a malfunction on the lower track, it blocks all oncoming traffic and might cause an unforeseen conflict. This means that the given information about induced follow-up conflicts isn't deterministically reliable.

7.7 Conclusion

In summary we have created a Reinforcement Learning environment to simulate railway operation and evaluate its quality from a passengers' perspective using Flatland. We have defined methods to carry out conflict detection and conflict resolution from a limited set of retiming and rerouting options. We have investigated the question whether deep networks trained by the Dueling Double DQN method can select conflict resolutions in a way that improves the quality of railway operation compared to heuristic baseline solutions. The training was done on randomly generated environments of different sizes and on a fixed grid, which consisted simply of two parallel tracks, both with and without Imitation and Curriculum Learning. The evaluation was also done on randomly generated environments of the same sizes as well as on the double track and on a realistic model grid with a realistic schedule. Delays were modelled to occur during railway operation following a negative exponential function.

So far the heuristic baseline solution achieves better quality than any of our trained networks, but the results indicate that Machine Learning approaches could close the gap with some more time and effort. However, the bad scaling doesn't bode well for the application of similar conflict resolution methods on larger railway networks. One possibility to avoid this scaling problem is to imitate the organisation of human train dispatchers, who operate only on local sections and need a global supervisor to coordinate the boundaries between those sections. In the same fashion railway operation on multiple small sections could be run in parallel and coordinated on a higher level at the boundaries.

Bibliography

[AGL20] Agasucci, Valerio ; Grani, Giorgio ; Lamorgese, Leonardo: Solving the single-track train scheduling problem via Deep Reinforcement Learning. In: *arXiv preprint arXiv:2009.00433* (2020)

[BLCW09] Bengio, Yoshua ; Louradour, Jérôme ; Collobert, Ronan ; Weston, Jason: Curriculum learning. In: *Proceedings of the 26th annual international conference on machine learning*, 2009, S. 41–48

[CCT08] Cacchiani, Valentina ; Caprara, Alberto ; Toth, Paolo: A column generation approach to train timetabling on a corridor. In: *4OR* 6 (2008), Nr. 2, S. 125–142

[CDH10] Corman, Francesco ; D'Ariano, Andrea ; Hansen, Ingo A.: Disruption handling in large railway networks. In: *WIT Transactions on the built environment* 114 (2010), S. 629–640

[CHK+14] Cacchiani, Valentina ; Huisman, Dennis ; Kidd, Martin ; Kroon, Leo ; Toth, Paolo ; Veelenturf, Lucas ; Wagenaar, Joris: An overview of recovery models and algorithms for real-time railway rescheduling. In: *Transportation Research Part B: Methodological* 63 (2014), S. 15–37

[CM20] Crespo Materna, Arturo: Dynamic and Intermodal Disruption-Management for Commuter Railway Networks. (2020)

[CMZ16] Cui, Yong ; Martin, Ullrich ; Zhao, Weiting: Calibration of disturbance parameters in railway operational simulation based on reinforcement learning. In: *Journal of Rail Transport Planning & Management* 6 (2016), Nr. 1, S. 1–12

[FYY15] Fang, Wei ; Yang, Shengxiang ; Yao, Xin: A survey on problem models and solution approaches to rescheduling in railway networks.

In: *IEEE Transactions on Intelligent Transportation Systems* 16 (2015), Nr. 6, S. 2997–3016

[HP14] Hansen, Ingo ; Pachl, Jörn: *Railway timetabling and operations: Analysis, modelling, optimisation, simulation, performance, evaluation.* Eurail press, 2014

[HQB+18] Horgan, Dan ; Quan, John ; Budden, David ; Barth-Maron, Gabriel ; Hessel, Matteo ; Van Hasselt, Hado ; Silver, David: Distributed prioritized experience replay. In: *arXiv preprint arXiv:1803.00933* (2018)

[Lat91] Latombe, Jean-Claude: Multiple moving objects. In: *Robot Motion Planning.* Springer, 1991, S. 356–402

[LCZ+21] Li, Jiaoyang ; Chen, Zhe ; Zheng, Yi ; Chan, Shao-Hung ; Harabor, Daniel ; Stuckey, Peter J. ; Ma, Hang ; Koenig, Sven: Scalable Rail Planning and Replanning: Winning the 2020 Flatland Challenge. In: *Proceedings of the International Conference on Automated Planning and Scheduling* Bd. 31, 2021, S. 477–485

[LSS+21] Laurent, Florian ; Schneider, Manuel ; Scheller, Christian ; Watson, Jeremy ; Li, Jiaoyang ; Chen, Zhe ; Zheng, Yi ; Chan, Shao-Hung ; Makhnev, Konstantin ; Svidchenko, Oleg u. a.: Flatland Competition 2020: MAPF and MARL for Efficient Train Coordination on a Grid World. In: *arXiv preprint arXiv:2103.16511* (2021)

[MBM+16] Mnih, Volodymyr ; Badia, Adria P. ; Mirza, Mehdi ; Graves, Alex ; Lillicrap, Timothy ; Harley, Tim ; Silver, David ; Kavukcuoglu, Koray: Asynchronous methods for deep reinforcement learning. In: *International conference on machine learning* PMLR, 2016, S. 1928–1937

[MKS+13] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Graves, Alex ; Antonoglou, Ioannis ; Wierstra, Daan ; Riedmiller, Martin: Playing atari with deep reinforcement learning. In: *arXiv preprint arXiv:1312.5602* (2013)

[MNL+20] Mohanty, Sharada ; Nygren, Erik ; Laurent, Florian ; Schneider, Manuel ; Scheller, Christian ; Bhattacharya, Nilabha ; Wat-

son, Jeremy ; Egli, Adrian ; Eichenberger, Christian ; Baumberger, Christian ; Vienken, Gereon ; Sturm, Irene ; Sartoretti, Guillaume ; Spigler, Giacomo: *Flatland-RL : Multi-Agent Reinforcement Learning on Trains*. 2020

[QCL15] Qu, Wenhua ; Corman, Francesco ; Lodewijks, Gabriel: A review of real time railway traffic management during disturbances. In: *International Conference on Computational Logistics* Springer, 2015, S. 658–672

[SB18] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement learning: An introduction*. MIT press, 2018

[Sch01] Schöbel, Anita: A model for the delay management problem based on mixed-integer-programming. In: *Electronic notes in theoretical computer science* 50 (2001), Nr. 1, S. 1–10

[Sil05] Silver, David: Cooperative pathfinding. In: *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment* Bd. 1, 2005, S. 117–122

[SKS+19] Sartoretti, Guillaume ; Kerr, Justin ; Shi, Yunfei ; Wagner, Glenn ; Kumar, TK S. ; Koenig, Sven ; Choset, Howie: Primal: Pathfinding via reinforcement and imitation multi-agent learning. In: *IEEE Robotics and Automation Letters* 4 (2019), Nr. 3, S. 2378–2385

[SKT18] Sato, Keisuke ; Koinuma, Keita ; Tomii, Norio: A train rescheduling algorithm which minimizes passengers' dissatisfaction based on MILP formulation. In: *CASPT2018-Conference on Advanced Systems in Public Transport and TransitData*, 2018

[SSF+19] Stern, Roni ; Sturtevant, Nathan R. ; Felner, Ariel ; Koenig, Sven ; Ma, Hang ; Walker, Thayne T. ; Li, Jiaoyang ; Atzmon, Dor ; Cohen, Liron ; Kumar, TK S. u. a.: Multi-agent pathfinding: Definitions, variants, and benchmarks. In: *Twelfth Annual Symposium on Combinatorial Search*, 2019

[SWD+17] Schulman, John ; Wolski, Filip ; Dhariwal, Prafulla ; Radford, Alec ; Klimov, Oleg: Proximal policy optimization algorithms. In: *arXiv preprint arXiv:1707.06347* (2017)

[TP05] Tornquist, Johanna ; Persson, Jan A.: Train traffic deviation handling using tabu search and simulated annealing. In: *Proceedings of the 38th annual Hawaii international conference on system sciences* IEEE, 2005, S. 73a–73a

[VHGS16] Van Hasselt, Hado ; Guez, Arthur ; Silver, David: Deep reinforcement learning with double q-learning. In: *Proceedings of the AAAI conference on artificial intelligence* Bd. 30, 2016

[WSH+16] Wang, Ziyu ; Schaul, Tom ; Hessel, Matteo ; Hasselt, Hado ; Lanctot, Marc ; Freitas, Nando: Dueling network architectures for deep reinforcement learning. In: *International conference on machine learning* PMLR, 2016, S. 1995–2003

[Yua06] Yuan, Jianxin: *Stochastic modelling of train delays and delay propagation in stations*. Bd. 2006. Eburon Uitgeverij BV, 2006

Appendix A

Python code and plots

We give an overview of all code that has been changed or added to the existing Flatland repository in order to make the functionalities of this thesis possible.

A.1 Adapted and new Python code

In the folder "envs"[1] changes have been made to the following files:

- rail_generators.py,
- line_generators.py,
- timetable_generators.py,
- malfunction_generators.py,
- rail_env.py.

In the folder "utils"[2] changes have been made to the following files:

- simple_rail.py,
- graphics_pil.py.

In the folder "step_utils"[3] changes have been made to the following files:

- malfunction_handler.py.

[1] https://gitlab.com/lcleicht/flatland/-/tree/master/flatland/envs
[2] https://gitlab.com/lcleicht/flatland/-/tree/master/flatland/utils
[3] https://gitlab.com/lcleicht/flatland/-/tree/master/flatland/envs/step_utils

In the folder "reinforcement_learning"[4] changes have been made to the following files:

- model.py,
- dddqn_policy.py.

In the folder "envs"[5] the following files were added:

- controllers.py,
- planning_controllers.py.

In the folder "reinforcement_learning"[6] the following files were added:

- multi_agent_test_env.py,
- multi_agent_random_env.py,
- Multi_Agent_Test_Env_Training.py,
- Multi_Agent_Random_Env_Training.py,
- Multi_Agent_All_Envs_Testing.py,
- Multi_Agent_Evaluation.py.

In the folder "notebooks"[7] the following Jupyter notebooks were added:

- CustomEnvironment.ipynb,
- DoubleTrack.ipynb,
- Multi_Agent_Test_Env_Training.ipynb,
- Multi_Agent_Test_Env_Testing.ipynb,
- Multi_Agent_Random_Env_Training.ipynb,
- Multi_Agent_All_Envs_Testing.ipynb,
- RandomEnvironment.ipynb.

[4] https://gitlab.com/lcleicht/flatland/-/tree/master/flatland_starter_kit_master/baselines/reinforcement_learning

[5] https://gitlab.com/lcleicht/flatland/-/tree/master/flatland/envs

[6] https://gitlab.com/lcleicht/flatland/-/tree/master/flatland_starter_kit_master/baselines/reinforcement_learning

[7] https://gitlab.com/lcleicht/flatland/-/tree/master/notebooks

A.2 Additional plots

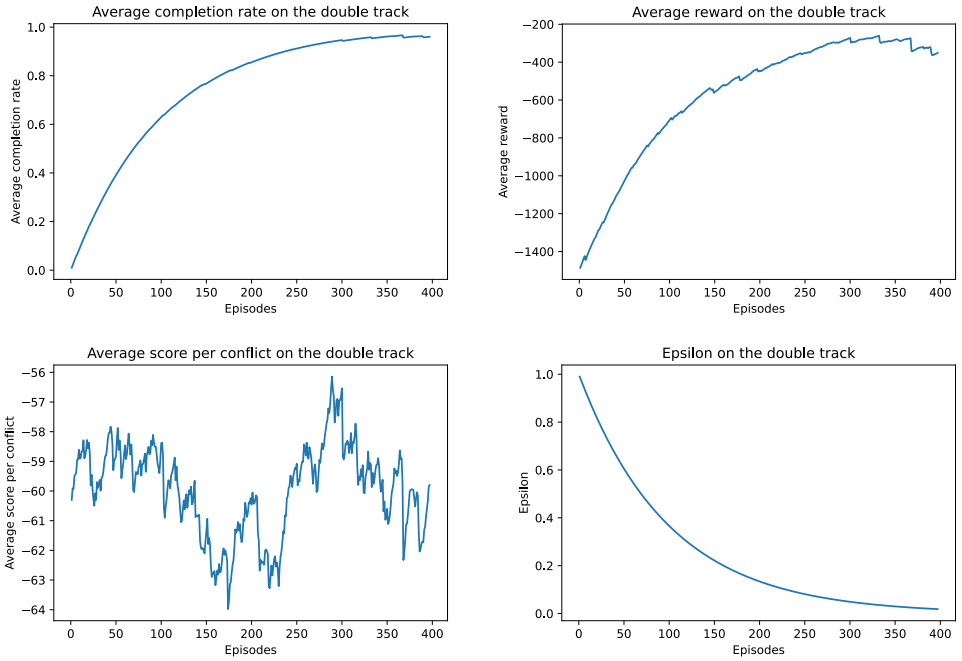


Figure A.1: The training process on the double track

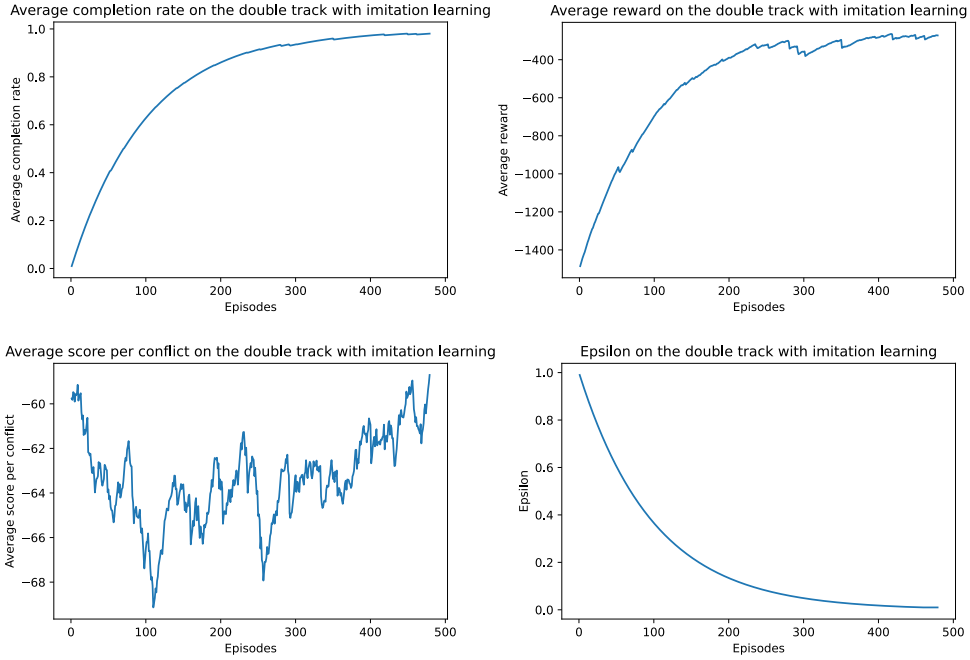


Figure A.2: The training process on the double track with imitation learning

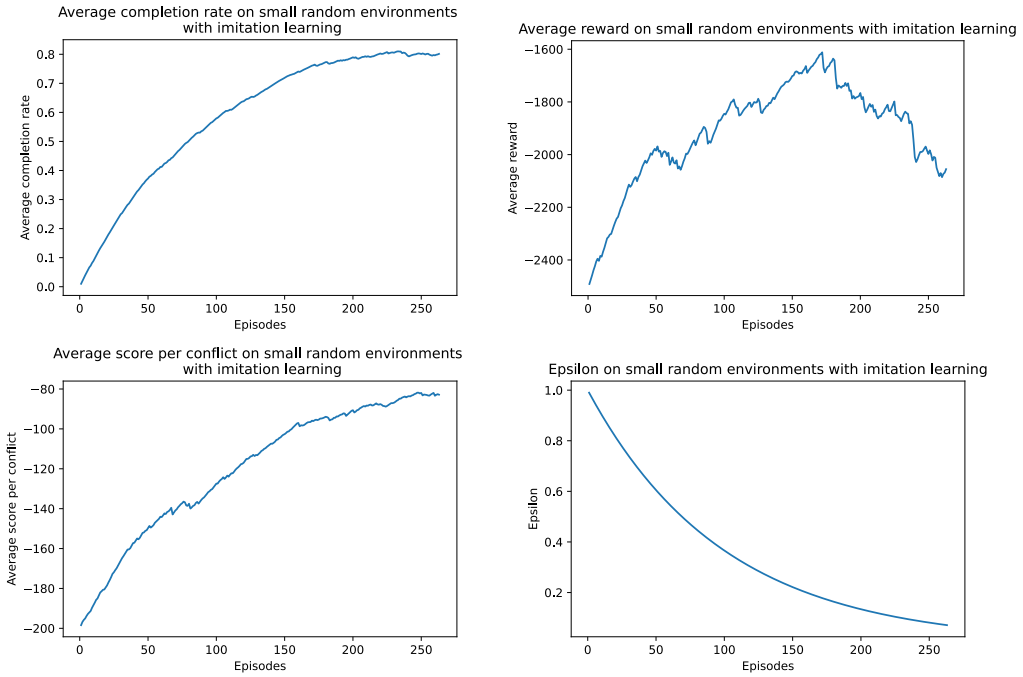


Figure A.3: The training process on small random environments with imitation learning

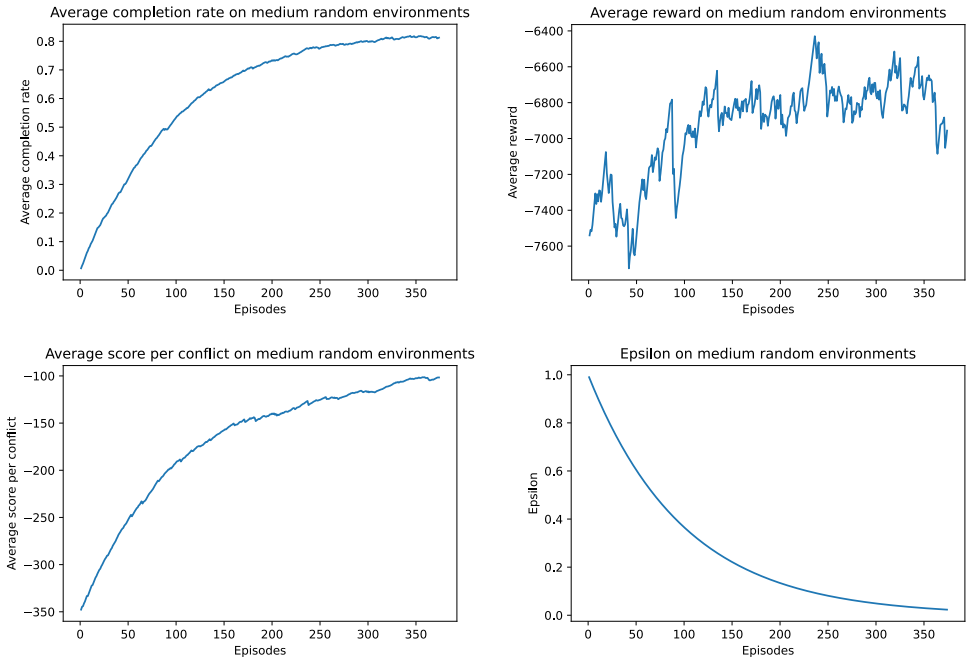


Figure A.4: The training process on medium random environments

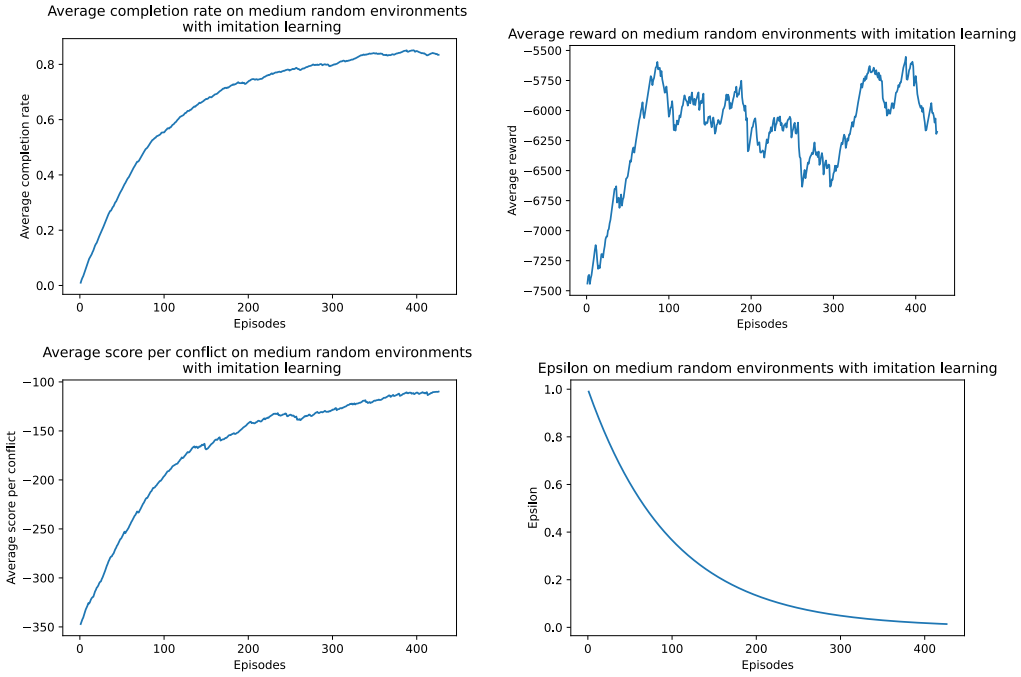


Figure A.5: The training process on medium random environments with imitation learning

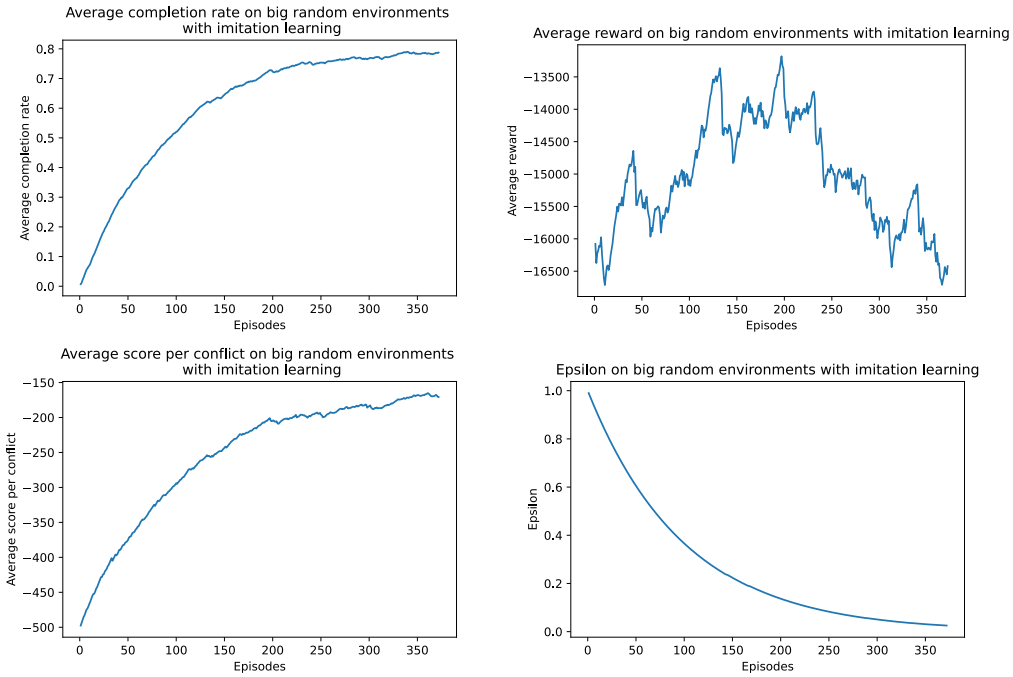


Figure A.6: The training process on big random environments with imitation learning

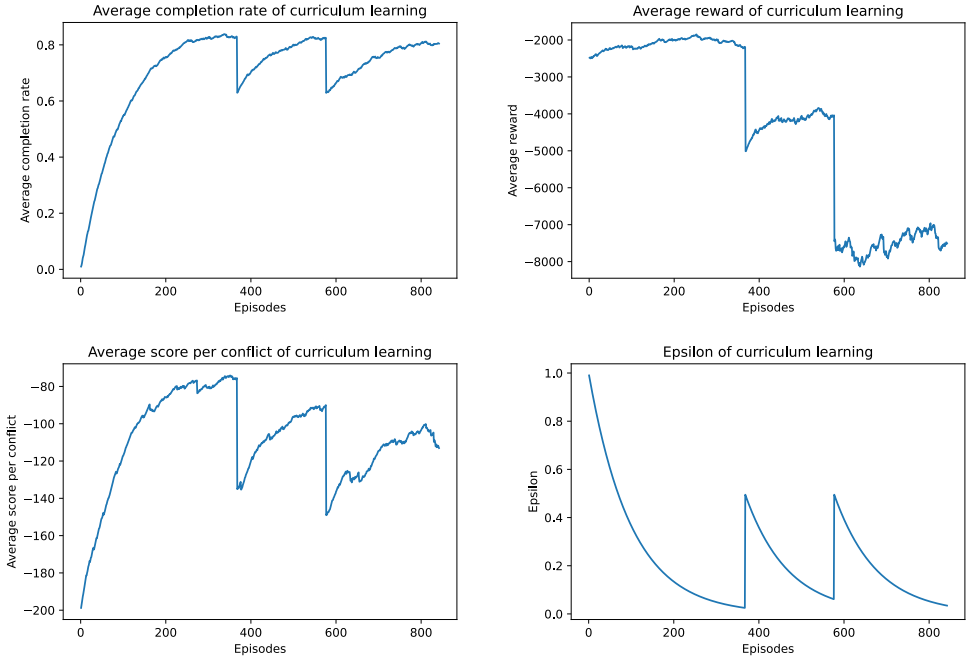


Figure A.7: The training process with curriculum learning

Index

Erklärung zur Abschlussarbeit gemäß §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Lorenz Leichthammer, die vorliegende Master-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Thesis Statement pursuant to §23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Lorenz Leichthammer, have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

In the submitted thesis the written copies and the electronic version for archiving are identical in content.

Frankfurt am Main, 31.03.2022

Lorenz Leichthammer