Exploring Reinforcement Learning Strategies for the Vehicle Rescheduling Problem

Amos Dinh
5504890
s212372@student.dhbw-mannheim.de

Matthias Fast
4750990
s212404@student.dhbw-mannheim.de

Henrik Rathai
7843611
s212387@student.dhbw-mannheim.de

Jannik Völker
5370226
s212416@student.dhbw-mannheim.de

***Abstract*— The Swiss Federal Railways manages one of the world's densest mixed railway network. With over 10,000 trains daily and more than 13,000 switches and 32,000 signals, traditional train scheduling methods often fail. The Flatland environment and associated challenges, created by the Swiss Federal Railways and AIcrowd, aim to improve train scheduling using a simplified 2D grid model for the Vehicle Rescheduling Problem. This paper focuses on Deep Reinforcement Learning approaches for solving the Vehicle Rescheduling Problem. We compare several algorithms including SARSA and Q-Learning. Our experiments show that all tested methods outperform the baseline shortest-path approach on the tested environment difficulties. The best-performing algorithms achieves a completion rate of up to 60% in the base environment, compared to the baseline's 15%. We also investigate the effects of observation depth, training length, and progressively increasing environment difficulty during training on performance. This work demonstrates the potential of Reinforcement Learning in addressing the Rescheduling Problem and provides suggestions for further research.**

***Index terms*—Multi-Agent RL, VRSP, Flatland3, Sarsa, Q-Learning, DDQN**

I. Introduction

The Swiss Federal Railways (SBB) operates and maintains the world's densest mixed railway traffic - Switzerland's largest railway infrastructure [1]. As of 2021, over 10,000 trains run daily, navigating through more than 13,000 switches and being controlled by over 32,000 signals [2]. In 2019, SBB aimed to increase network transportation capacity by about 30% [3].

Therefore, the Flatland environment organized by the Swiss Federal Railways (SBB) and the AIcrowd platform with the partners Deutsche Bahn (DB) and Société Nationale des Chemins de Fer Français (SNCF) has emerged to meet rising mobility demands. This environment has hosted several challenges to date, including the AMLD 2019 Flatland Challenge [3], the NeurIPS 2020 Challenge [4], the AMLD 2021 Challenge [4], and the Flatland 3 Challenge [5]. Each challenge features slightly different environment configurations and enhancements designed to improve train scheduling and rescheduling processes.

At the core of the Flatland Challenge lies the general vehicle rescheduling problem (VRSP):

> *"The vehicle rescheduling problem (VRSP) arises when a previously assigned trip is disrupted. A traffic accident, a medical emergency, or a breakdown of a vehicle are examples of possible disruptions that demand the rescheduling of vehicle trips. The VRSP can be approached as a dynamic version of the classical vehicle scheduling problem (VSP) where assignments are generated dynamically"*
>
> — [6]

The VRSP is NP-complete, making it complex when accounting for all real-world factors in SBB's advanced simulation [6]. To speed up innovation, the Flatland environment was introduced as a simplified 2D grid model [2].

A. Paper overview

The next section, Environment, provides an overview of the Flatland 3.0 environment for VRSP, including its grid layout and agent behaviors. Following this, in Methodology, we outline the problem framing and describe the Reinforcement Learning algorithms employed along with the experimental settings and evaluation criteria. Afterwards, in the Results and Analysis section, we present our baseline performance and compare the different RL approaches, focussing on the effects of training length, progressively harder envi-

ronments and observation depth. Finally, we summarize the key findings and propose directions for future research.

II. Environment

This chapter provides a description of the Flatland 3.0 environment as detailed in [2].

The Flatland environment is simulated as a rectangular 2D grid of dimension W x H. Each cell in the grid is either a target, a rail cell or an empty unusable cell. Rails vary in type based on the available transitions: there are 16 different transitions in Flatland, given that there are 4 orientations and 4 exit directions from each cell. To simulate a realistic railway system, only a maximum of 2 exit directions are allowed for each orientation of the agent. This constraint results in 8 distinct cell types, visualized in Figure 1. Each rail cell can be rotated by 90 degrees and flipped horizontally or vertically.

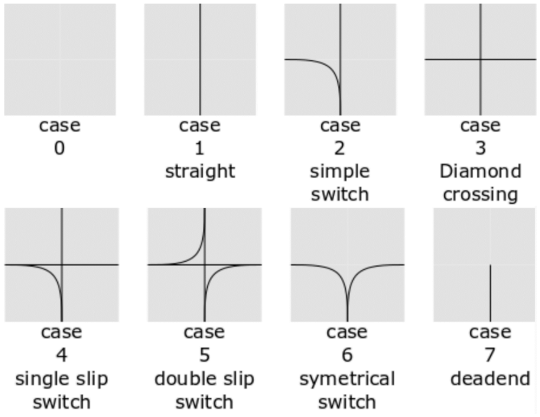

Figure 1: Types of rail configurations, including straight tracks, various switches, crossings and a deadend [3]

All outgoing connections of a cell must align to the incoming connections of an adjacent cell. These rules are applied to generate random and valid railway network configurations which are then used for training purposes.

Rails can host agents and targets, with each rail holding only one agent at a time, as shown in Figure 2 on the left. The target, depicted in Figure 2b, is randomly assigned to a rail. Target cells serve as destinations for agents, and multiple agents can share the same target. The number of targets is limited by the number of agents.

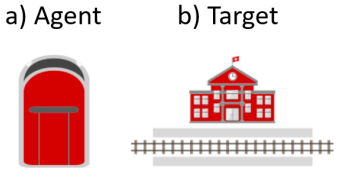

Figure 2: Agents and Targets in Flatland [3]

The Flatland Environment includes a sparse_rail_generator that creates realistic railway layouts by randomly connecting clusters of cities in a sparse manner. An example can be seen in Figure 3.

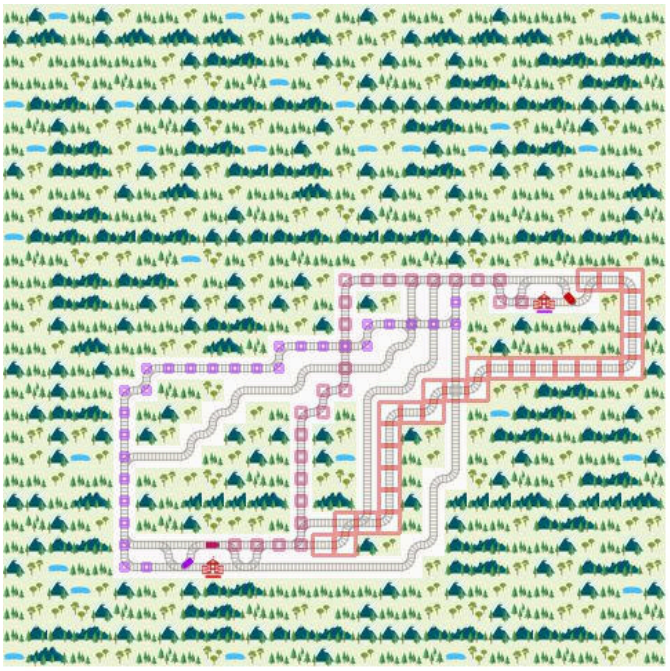

Figure 3: Generated railway network with three agents and two targets. For each agent, the path to its respective target is visualized with a maximum of 30 steps.

A Rail segment is encoded as 4×4 (flattened) binary matrix where an entry at i, j indicates whether an agent traveling towards cardinal direction i can leave the cell in cardinal direction j. This complicated notation is necessary, since for rail intersections, unlike road intersections, vehicles most often can not leave towards an arbitrary direction from which it is possible to enter the intersection.

If an agent encounters a rail type with multiple direction options, it must choose a path. The action space includes:

1. MOVE_FORWARD: Continue in the current direction if possible.
2. MOVE_LEFT: Take the left path at a switch if available; otherwise, no effect.
3. MOVE_RIGHT: Take the right path at a switch if available; otherwise, no effect.
4. STOP_MOVING: Remain in the current cell.
5. DO_NOTHING: Repeat the action from the last step (default action).

A. Complications

In simulating a realistic railway system, several complications arise that must be addressed. Agents can operate at different speeds, a fraction between 0 and 1, representing different types of trains, where 1 denotes one cell per time step.

Each agent, similar to real trains, can experience malfunctions. Key properties such as the malfunction rate (probability per time step) and duration of a malfunction can be configured in the environment generation. Up until version 2 of Flatland trains could depart and arrive at any time. However, real-world railways prioritize timing and punctuality, with trains adhering to specific schedules. Since Flatland 3 trains have a designated time window for departure and arrival. Due to increased complexity the setting with time tables and speed profiles is excluded in the present experiments.

B. Observations

Figure 4 visualizes the different types of observations provided in the flatland environment.

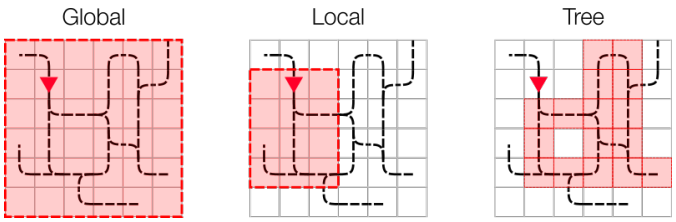

Figure 4: Types of observations provided in the flatland environment: global, local and tree observation [7]

The global observation provides a complete view of the entire Flatland environment for each agent. It includes transition maps, agent states, and agent targets [7].

The local grid observation offers a localized view centered on the agent, rotated according to the agent's direction. It includes a distance map to provide direction without the need for a global view, though it is deprecated due to limited results [7].

The tree observation creates a tree along the paths the agent can take. Each node in the tree represents a crossing and has information about targets, other agents, and potential problems, based on the agent's direction [7], such as the distance to the respective objects or the direction of agents.

III. Methodology

A. Problem Framing

The experiments aim at examining different Reinforcement Learning approaches and hyper-parameter configurations on the described problem setting. Given observation of the environment at each time step, an algorithm should determine for each agent the best sequence of actions to take, in order to lead each agent to their target train station in minimal time. During the simulation, agents (the trains) have to avoid other malfunctioning trains.

We frame the task as a single-agent problem (not a multi-agent one). Although we use one single algorithm to steer each train (agent) individually, the reward is formulated as a accumulation of the individual agents rewards. Thus it amounts to a single objective - not multiple opposing ones per train and can be interpreted as one agent making multiple decisions per time step. For such a unified reward formulation, it could be in the agent's interest to pause one train temporarily in order to allow multiple other trains to reach their targets faster.

B. Reinforcement Learning Approach

We compare multiple Reinforcement Learning algorithms to learn a suitable policy. Among them, some can be categorized in *off-policy* and *on-policy*, meaning the action distribution of agents during training is separate from the learned optimal policy in the former case. We denote A as the action, S as the state, t as the timestep, R as the reward and $Q(A, S)$ as the state-action function. The agents learn to estimate the Q-value and thereby allow to estimate the best action given a state. In detail, one-step temporal difference learning is applied, where we update $Q(S_t, A_t)$ based on $Q(S_{t+1})$ and the reward R_{t+1} which resulted from taking action A_t.

Further, Experience Replay is utilized [8]. Here, past experiences are stored in a queue with limited size. For re-estimation of the Q-values, past experiences which are learned from are sampled uniformly at random. This approach may help to "*smooth[...] the training distribution*" [9] by allowing the updating based not solely on the most recent actions.

For the Reinforcement Learning algorithms, an ε-greedy approach is used to initially encourage exploration of the state space. Here the probability of taking a random action is defined as $\max(0.99^t, 0.01)$.

We employ a deep learning based backbone to estimate Q, meaning Q(S,A) = MLP(S,A) where MLP is a Multi Layer Perceptron with two hidden layers. The training follows the approach of [10], where the loss L is defined as

$$L = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right)^2\right] \quad (1)$$

The network parameters are soft-updated after a gradient step (Implementation already provided by the flatland environment) [11]:

$$\theta' \leftarrow \rho\theta + (1 - \rho)\theta' \quad (2)$$

The Reinforcement Learning algorithms utilized are presented now. We use the definition below to shorten the following equations:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \delta \quad (3)$$

Here δ represents the difference in the estimate to update $Q(S_t, A_t)$ with.

1) *Shortest Path:* This algorithm always chooses the action corresponding to the shortest path to the destination for each agent and represents the baseline. The algorithm is provided by the library it-self and uses a breath-first search to create a distance map for each railroad segment respective to each target destination.

2) *SARSA (on-policy):*

SARSA [12] estimates the action-value function Q for all states and actions. The update rule is:

$$\delta = \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{4}$$

where α is the learning rate, γ is the discount factor. For terminal states, $Q(S_{t+1}, A_{t+1})$ is defined as zero. We can utilize the loss formulation in Equation 1 to learn the deep learning-based Q, replacing $\max_{a'} Q(s', a')$ with $Q(s', a')$ where a' is the real action taken in the scenario.

The one-step TD can be extended to the n-step scenario:

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)] \tag{5}$$

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}) \tag{6}$$

3) *Q-Learning (off-policy):*

For Q-Learning, the update rule is defined as:

$$\delta = \alpha\Big[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)\Big] \tag{7}$$

with the corresponding loss function from Equation 1.

The Q-Learning update is executed after each transition. For terminal states S_{t+1},$Q(S_{t+1}, A_{t+1})$ is set to zero.

Q-Learning is categorized as an off-policy method because it evaluates the optimal action

$$\max_a Q(S_{t+1}, a) \tag{8}$$

for the next state, rather than the action actually taken in the next state, distinguishing it from the on-policy nature of SARSA. This allows Q-Learning to potentially learn an optimal policy independently of the agent's actions. [13]

4) *Double Q-Learning (off-policy):*

In double Q-Learning [14], [15], two estimators for Q are employed in tandem to reduce the probability of over-estimation of the next state-action pair's Q-value. $\max_a Q(S_{t+1})$ in Equation 4 is replaced with $Q_2(S_{t+1}, \max_a Q_1(S_{t+1}, a))$, which means that the Q-value to select the optimal action and the value used of the update are distinct from each other.

5) *Double Dueling Q-Learning (off-policy):*

Double Dueling Q-Learning [16] extends the concept of Double Q-Learning by integrating the Dueling Network Architecture, which separates the estimation of state value and advantage functions. An implementation of this method is already provided by the Flatland challenge.

Dueling Network Architecture:

The value function $V(s)$ estimates the value of being in a state s. The advantage function $A(s, a)$ estimates the relative value of performing action a in state s. The combined Q-value is computed as:

$$Q(s, a) = V(s) + (A(s, a) - \mathbb{E}_{a'}[A(s, a')]) \tag{9}$$

Double Dueling Q-Learning (off-policy): Similar to Double Q-Learning, two sets of value functions Q_1 and Q_2 are maintained. For action selection, the dueling network is used to select the best action a' using Q_1. The selected action is then evaluated using Q_2. The update rule for Q_1 in Double Dueling Q-Learning is:

$$\delta = \alpha\Big[R_t + \gamma Q_2\Big(S_{t+1}, \arg\max_{a'} Q_1(S_{t+1}, a')\Big) - Q_2(S_t, A_t)\Big] \tag{10}$$

The same procedure is applied to update Q_2 using Q_1.

6) *Expected-SARSA:*

Lastly, we experiment with the Expected-SARSA approach (11) and define π as the Boltzman-Statistic [17] with hyperparameter τ (12).

$$\delta = \alpha[R_t + \gamma Q_2\Big(S_{t+1}, \sum_{a'} \pi(a'|S_{t+1}) Q(S_{t+1}, a') - Q(S_t, A_t)\Big] \tag{11}$$

$$\pi(a'|S_{t+1}) = \left(\sum_a e^{\frac{Q(S_{t+1}, a)}{\tau}}\right)^{-1} e^{\frac{Q(S_{t+1}, a')}{\tau}} \tag{12}$$

C. Experimental Settings

In the experiments, the tree observation method is used due to smaller state size and the reduced computational requirements compared to the global view and higher infor-

mation density compared to the local view. The challenge already provides the code to construct this observation type.

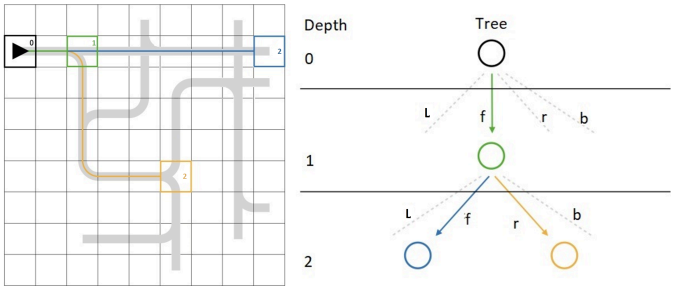

Figure 5: Tree observation, depth 2 [2]

The observation for depth 2 is constructed as shown in Figure 5. The root node is the current position of the agent and each depth adds 4 nodes per parent node (forwad, right, backward and left). The following nodes represent intersections where agents can make decisions. Each node contains twelve features such as "min_dist_to_tagrget", "target_on_node", "other_agent_on_node", "other_agent_on_node_at_timestep_t". To predict other agent's positions at following time steps a shortest-path predictor is used which predicts the other agent's positions at later time steps based on their shortest path to their target when ignoring all other agents. More sophisticated predictors could improve the agent's planning ability.

1) *Evaluation Criteria:*

We use the agents training performance as evaluation performance, since environments are generated with different layouts every episode. All generated levels mostly share the following specifications:

- speed = 1.0
- max_rails_between_cities = 2
- max_rails_in_city = 2
- n_envs_run =2000

All trains travel at the same speed. Furthermore, there can be a maximum of two rails between cities and a maximum of two rails within cities.

For each level, the environment increases in size. The parameters are listed in Table 1.

env	#agents	w	h	#cities	malfunction rate
0	7	30	30	2	0
1	10	30	30	2	0.010
2	20	30	30	3	0.005
3	50	30	35	3	0.005

Table 1: Parameter configuration for environment versions

2) *Rewards:*

We utilize the default rewards provided by the environment. Defining an appropriate reward structure to achieve the target goal could be subject to further investigation. The reward structure is as follows for each agent:

- A negative reward for not moving at all
- A negative reward penalty per time step if it has not yet reached its destination.
- A positive reward if it has reached its destination.

The total returns are the rewards the agents gather throughout each episode. To allow for comparison between environments of different sizes the returns are normalized to a scale between -1 and 0:

```
normalized_reward =
(cumulative_reward / (self.env._max_episode_steps *
self.env.get_num_agents()))
```

IV. Results and Analysis

Here a number of experiments are presented which compare the different reinforcement learning approaches as well as the performance of different hyperparameter configurations.

A. Baseline

As a baseline, we instruct the agent to always choose the shortest path. The smoothed score hovers around -0.5 for environments 0, 1, and 2. The smoothed score can be seen in Figure 6.

A more intuitive measure is the completion rate, denoting the fraction of trains reaching their destination within one episode.

In some episodes, the baseline reaches a completion rate of 100% in environment 0, meaning that all trains reach their target. The baseline achieves its highest completion rate in the least complex environment 0 at 15%. The completion rate in environments 1 and 2 reach 8% and 6% respectively as shown in Figure 7.

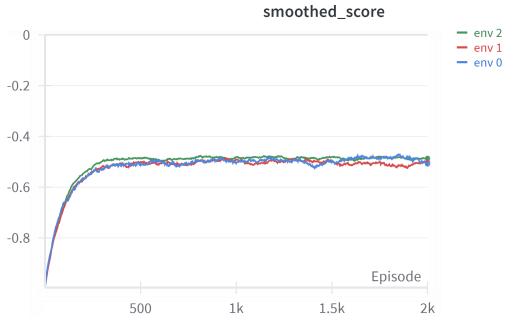

Figure 6: Smoothed score of the baseline shortest-path policy comparing the three evaluation environments 0, 1, and 2 over 2000 episodes

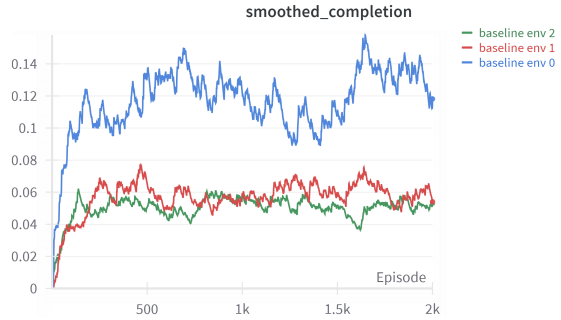

Figure 7: Completion rate of the baseline denoting the number of agents reaching their destination during an episode comparing the three evaluation environments 0, 1, and 2 over 2000 episodes. In env 0 14% of the trains reach their destination.

B. Comparison of different Reinforcement Learning Algorithms

We train the policies SARSA, DQN, Dueling DQN, Double DQN, and Double Dueling DQN on 2000 episodes, trying three different training environment configurations. We use environments 0, 1, and 2 as seen in Table 1.

It can be observed that the evaluation of the models yields the highest score on environment 0 since it offers the least complexity in terms of the number of agents and the absence malfunctions. Comparably, the used policy itself has a smaller effect on the score. These findings are visualized in Figure 8.

Figure 9 shows the convergence of the training scores. Comparing these results with our baseline reveals that every policy performs better than the baseline. On Environment 0 the biggest improvement over the baseline is achieved with a completion rate of up to 60% compared to the baseline's 15%

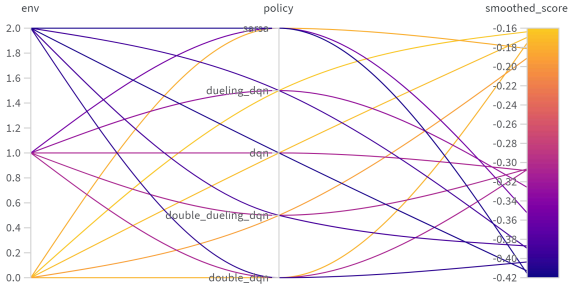

Figure 8: Comparison of policies and environments affecting the score where each line represents one training run. The enviroment difficulty has greater influence on the performance. Different algorithms perform similarly.

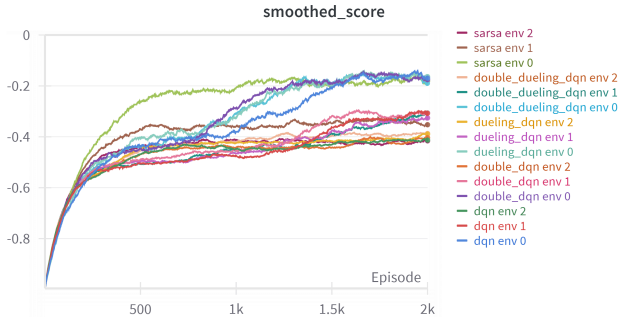

Figure 9: Smoothed score for the three evaluation environments 0, 1, and 2 over 2000 episodes comparing SARSA, DQN, Dueling DQN, Double DQN, and Double Dueling DQN. SARSA learns quicker but the Double Dueling DQN can achieve slightly higher performance.

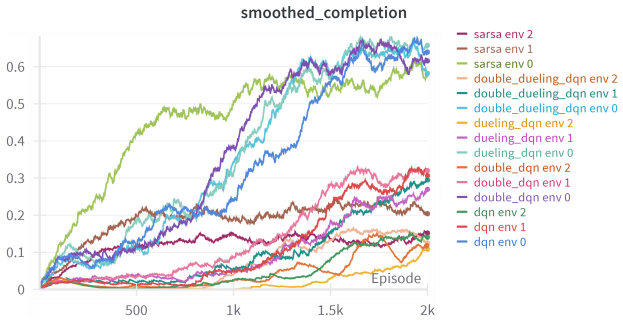

Figure 10: Completion rate for the three evaluation environments 0, 1, and 2 over 2000 episodes comparing SARSA, DQN, Dueling DQN, Double DQN, and Double Dueling DQN.

During training, it can be observed that all policies initially start with a probability of 20 % for the five different actions and then quickly increase the probability for "stopping" and "doing nothing" while decreasing the probability for the other three actions. However, these differences decrease as the training progresses. The training progression is shown in Figure 11.

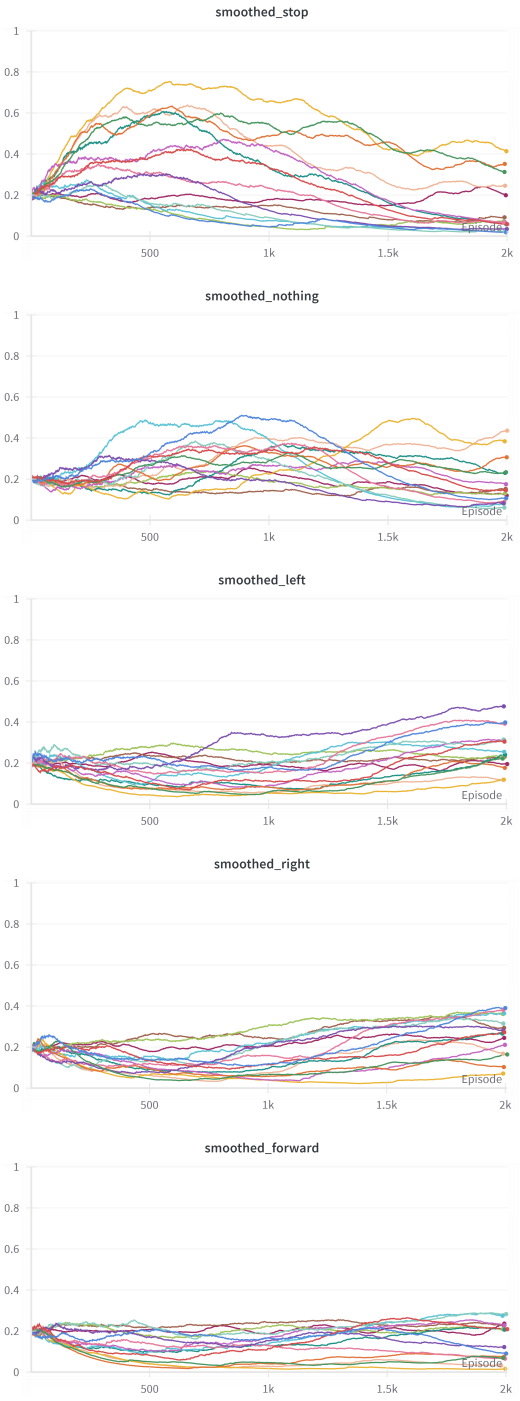

Figure 11: Probability of a given action comparing different policies and environments over 2000 episodes. Initially the five actions start out with a probability of 20%. The “stop” probability and “nothing” probabilties increase in favor of the other actions’ probabilities

C. *Length of Training*

We train the Double DQN and Double Dueling DQN for 10000 episodes to check whether our assumption of convergence after 2000 episodes holds true. Indeed, Figure 12 shows that after 2000 episodes there is no further improvement in the score for the two tested methods. Therefore we deem the limit of 2000 episodes as sufficient for the majority of our experiments.

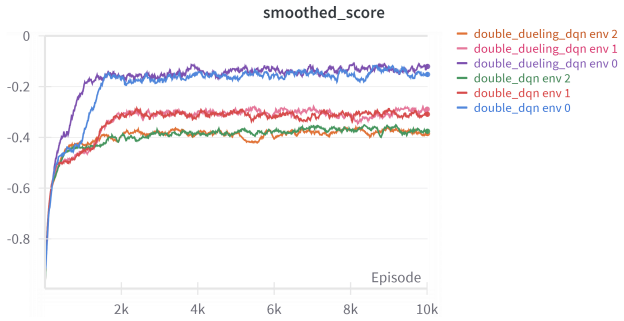

Figure 12: Smoothed score over 10000 episodes for the three evaluation environments 0, 1, and 2 comparing DQN and Double Dueling DQN

D. *Training on progressively harder environments*

Another aspect we examine is progressively increasing the environment’s difficulty during training. The assumption is that a policy trained on the most difficult environment from the start may tend to be unstable because it can not handle the complexity. Therefore, gradual increasing complexity might help the algorithm to first establish basic concepts to later focus on the more complex settings. We try two different approaches. First, we train a policy for 2000 episodes on one environment, then switch to the next harder level and train for 2000 episodes again. The results of this stacked approach can be seen in Figure 13. It can be observed that the completion rate decreases drastically after each environment change but sometimes recovers again (Such as for the double_dqn at step 5000).

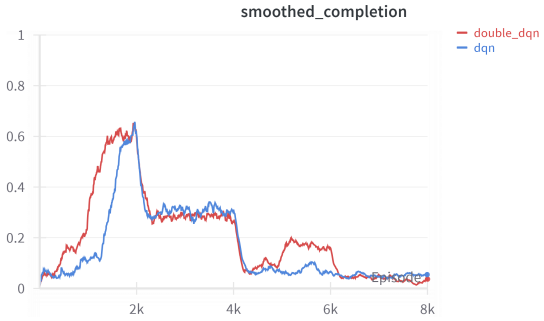

Figure 13: Smoothed completion rate for the stacked approach over 8000 episodes wich increases the training environments complexity every 2000 episodes.

We also try to increase the agent count and environment size in increments of one, training a policy for 20,000 episodes using this approach. The completion rate increases for about 5,000 episodes and then decreases again. The final completion rate is slightly above the previous experiment’s, indicat-

ing that this approach might be more promising but requires further adjustments to scale beyond that threshold, as seen in Figure 14.

With more abundant computational resources, exploring the setting of training with increasing complexity in conjunction with different ε-greedy schedules and queue sizes of the replay buffer could yield better results.

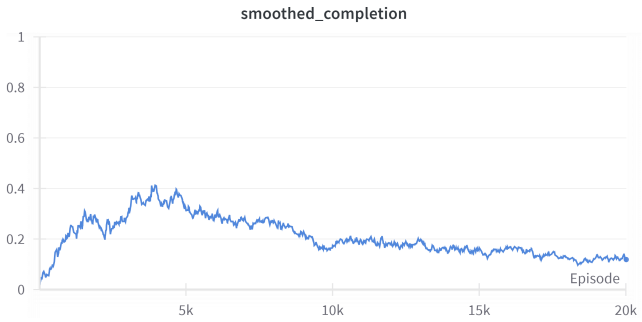

Figure 14: Smoothed completion rate for the iterative approach over 20000 episodes increasing environment size and agent count iteratively. Adding trains decreases the algorithms performance.

E. *Observation Depth*

As described in Section III.C the depth of the observation tree can be adjusted. How this affects the score can be seen in Figure 15. There is almost no effect on the score when changing the observation tree depth. Most of the difference in the score is again due to the three training environments

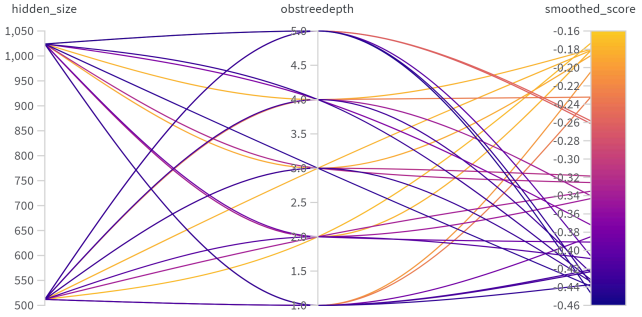

Figure 15: Effect of the hidden size and depth of the observation tree on the final score where each line represents one training run, including all three environments

F. *N-Step SARSA*

We compare different number of steps for the n-step SARSA policy as well as two different discount factors γ. Figure 16 shows, that small numbers of steps affect the score positively. γ has no large influence on the score.

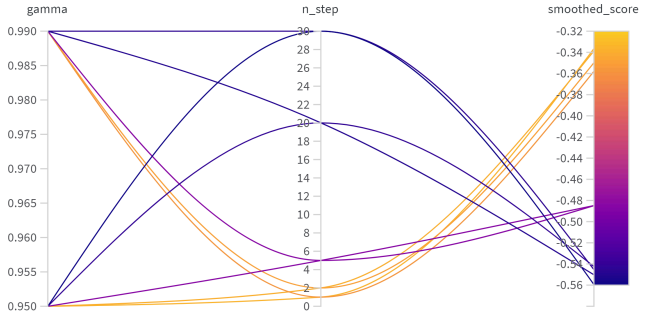

Figure 16: Comparison of gamma and the step size affecting the score for n-step SARSA where each line represents one training run in environment 1. Gamma has no large effect on the score but smaller step sizes are beneficial.

G. *Expected Sarsa*

Similarly we compare the temperature for the expected SARSA Policy. There doesn't seem to be clear link between temperature and score. This is illustrated in Figure 17.

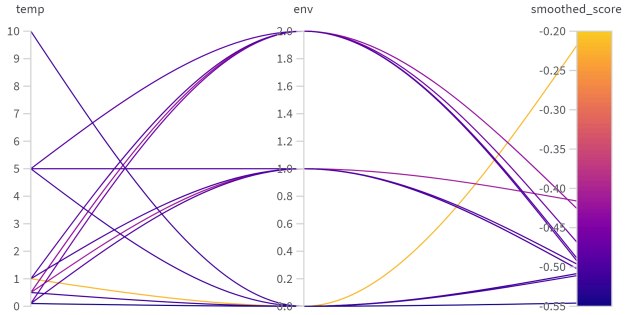

Figure 17: τ has no large effect on the score for Expected-SARSA where each line represents one training run

H. *Generalization ability*

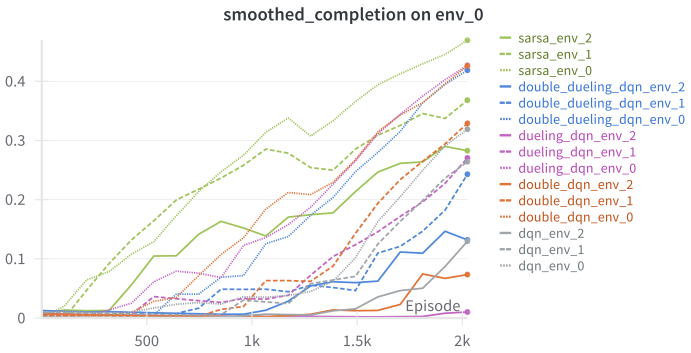

Figure 18: Trained policies evaluated on environment 0. Training Environments are given in the legend. SARSA has the best generalization ability.

We further examine the generalization ability by testing the policies trained on different environment difficulties on the simplest environment 0 , as seen in Figure 18. SARSA has the highest generalization ability. Interestingly, the policies trained on environment 2 perform better than those trained on environment 0 itself. The policies of environment 0 seemingly perform worse than in the training evaluation in Figure 10 which is caused by the smoothing of scores combined with the lower sampling frequency in this evaluation run.

I. Changing the observation

A brief experiment was conducted, exchanging the tree-based observation with a graph based one. A graph is constructed where each rail-segment is represented by a node. Neighboring segments are connected by edges and sequences of straight rails are collapsed into single nodes. Features are the transition matrix as well as features similar to the tree-based observation. We use a Graph Neural Network as estimator for Q. Unfortunately, the performance of the method with small adjustments is worse than the baseline and we suspect that the graph was missing important spatial information for the Graph Neural Network to distinguish neighboring nodes from each other.

V. Conclusion

This paper demonstrates how the Reinforcement Learning algorithms can outperform the baseline shortest-path approach in solving the Vehicle Rescheduling Problem (VRSP) in the Flatland environment. By comparing the Reinforcement Learning algorithms SARSA and Q-Learning and their derivatives expected Sarsa, n-step SARSA, DQN, Double DQN, Dueling DQN, and Double Dueling DQN, we find that all methods can outperform the shortest path strategy in our testing environment.

Our results show improved scores and also completion rates compared to the baseline. We also show that the training length is only effective until a certain point with the utilized network architecture.

We discover that the observation depth in general, as well as the temperature for expected SARSA is not indicative of the score, but keeping the number of steps for n-step SARSA low would be beneficial.

Our findings are limited in scope. Therefore, claims made can only be considered in the provided context. For example, a more sophisticated predictor which is utilized in the construction of the tree observation could allow the trains to "communicate" with each other more efficiently, avoiding deadlocks. For such a setting, the influence of hyperparameters could change.

Promising directions for further research could be:

- Increasing the environment's difficulty progressively during training, since we find that it could increase the score.
- During the experiments we found the number one cause of difficulty to be deadlocks, where the agents would block each others actions, due to frontal or sideways "collisions" as well as the malfunctions. This could be caused by the lack of information provided by the observation or the insufficiency of the standard predictor which only inaccurately predicts the future position of other trains.

References

[1] "SBB Statistikportal." [Online]. Available: https://reporting.sbb.ch/

[2] S. Mohanty *et al.*, "Flatland-RL : Multi-Agent Reinforcement Learning on Trains." [Online]. Available: https://arxiv.org/abs/2012.05893

[3] "AIcrowd." [Online]. Available: https://www.aicrowd.com/challenges/flatland-challenge

[4] "AIcrowd." [Online]. Available: https://www.aicrowd.com/challenges/flatland

[5] "AIcrowd." [Online]. Available: https://www.aicrowd.com/challenges/flatland-3

[6] J.-Q. Li, P. B. Mirchandani, and D. Borenstein, "The vehicle rescheduling problem: Model and algorithms," *Networks*, vol. 50, no. 3, pp. 211–229, 2007, doi: https://doi.org/10.1002/net.20199.

[7] "AIcrowd." [Online]. Available: https://flatland.aicrowd.com/environment/observations.html#local-grid-observation

[8] L.-J. Lin, *Reinforcement learning for robots using neural networks*. Carnegie Mellon University, 1992.

[9] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[10] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[11] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[12] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*, vol. 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[13] C. WATKINS, "Learning from Delayed Rewards," *PhD thesis, Cambridge University, Cambridge, England*, 1989.

[14] H. Hasselt, "Double Q-learning," in *Advances in Neural Information Processing Systems*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds., Curran Associates, Inc., 2010, p. . [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf

[15] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning." [Online]. Available: https://arxiv.org/abs/1509.06461

[16] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling Network Architectures for Deep Reinforcement Learning." [Online]. Available: https://arxiv.org/abs/1511.06581

[17] L. Boltzmann, *Studien über das Gleichgewicht der lebendigen Kraft zwischen bewegten materiellen Punkten: vorgelegt in der Sitzung am 8. October 1868*. k. und k. Hof-und Staatsdr., 1868.