Multi-Agent Reinforcement Learning in a Railway Traffic Management System

Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg East, Denmark
www.cs.aau.dk

Title

Multi-Agent Reinforcement Learning
in a Railway Traffic Management System

Project type

7^{th} Semester Project:
Secure, Scalable and Useful Systems

Project period

Fall 2021

Project group

4

Participants

Dennis Tran
Jakob Rønnest Sørensen
Kim Nguyen
Nicolai Harbo Christensen
Simon Eliasen
Thomas Vinther

Supervisor

Alvaro Torralba

Abstract

An automated Traffic Management System (TMS) is a prospective solution that can address the increase in complexity and efficiently manage railway traffic. As such, this project investigates the development of a Reinforcement Learning (RL) based TMS for a simplified 2D grid environment named Flatland.

The developed TMS utilizes an actor-critic Proximal Policy Optimization (PPO) architecture, along with a customized tree-based observation model. Furthermore, the developed TMS has been optimized with curriculum learning, frame skipping and hyperparameter tuning.

Overall the developed TMS shows potential regarding scalability, adaptability, and generalizability, but in its current state, it is not applicable in a real railway network. Nevertheless, within the confines of the Flatland challenge 3, it achieved a third place in the RL submission category.

Number of pages: 96
Date of completion: January 25, 2022
Front page image credits: Flatland, 2021

Preface

This semester project was written by a 7^{th} semester Computer Science project group at Aalborg University.

References follow the Harvard referencing standard. Further, reference information can be found in the bibliography provided in the end. The bibliography is sorted alphabetically by author. However, if the author is unknown, the publisher is used instead. If the reference is a website, the latest visiting date is also included.

Figures and tables are sorted according to the chapters. Every figure found externally will have a source reference. Figures without a reference are created by the authors themselves.

Numbers are written using periods as decimal separators and spaces as thousand separators.

All source code is available in the following repository:
https://github.com/simoneliasen/Flatland-MARL-TMS

The authors of this project are:

Dennis Tran	Jakob Rønnest Sørensen	Kim Nguyen
Nicolai Harbo Christensen	Simon Eliasen	Thomas Vinther

Contents

Nomenclature

Abbreviations

SBB	*Swiss Federal Railway Company*
VRSP	*Vehicle Rescheduling Problem*
TMS	*Traffic Management System*
ML	*Machine Learning*
RL	*Reinforcement Learning*
OR	*Operations Research*
MAPF	*Multi-Agent Pathfinding*
PPO	*Proximal Policy Optimization*
IL	*Imitation Learning*
MDP	*Markov Decision Process*
MLP	*Multilayer Perceptron*
Tanh	*Hyperbolic Tangent*
ReLU	*Rectifier Linear Unit*
MLE	*Maximum Likelihood Estimate*
TD	*Temporal Difference*
GAE	*Generalized Advantage Estimation*
DDQN	*Double Dueling Q-Network*
TRPO	*Trust-Region Policy Optimization*
MARL	*Multi-Agent Reinforcement Learning*
MAPPO	*Multi-Agent Proximal Policy Optimization*
CL	*Concatenations of Local Observation*
EP	*Environment-Provided Global State*
AS	*Agent-Specific Global State*
EMA	*Exponential Moving Average*
RNN	*Recurrent Neural Network*

Mathematical Notation

a	A scalar (integer or real)
$\boldsymbol{a}$	A vector
$\mathbb{A}$	A set

Introduction 1

Along with societal progression and technological evolution, transport activity across Europe continues to grow. The European Union estimates the transport for passengers will increase by 42 % by 2050, and freight transport by 60 %. This puts heavy pressure on the transport network, making it necessary for transportation of passengers and goods to be efficient. This capacity crunch is already being felt across sectors in the EU [EC, 2019, p. 1].

For instance, the railway traffic network is heavily affected by the increase in transportation density. The Swiss Federal Railway Company (SBB) operates the densest mixed railway traffic network in the world, carrying both people and goods. They have forecast the need for an increase in transportation capacity by approximately 30% in the near future [Mohanty et al., 2020, p. 2].

The demands of the future needs to be meet with an increase in transport capacity. In this regard, a key component which needs to be addressed and handled are disruptions. Common causes for traffic congestion across all transportation sectors are through disruptions, such as when a scheduled vehicle breaks down or is involved in a collision en route. Such instances, demand efficient rescheduling of vehicles. Otherwise, disruptions can have a significant impact on the stability of a transport network. The question then arises of how to reschedule the vehicles, which is known as the Vehicle Rescheduling Problem (VRSP). VRSP can be related across all logistics and transportation sectors: Road, railway, airlines, and marines [Li et al., 2007, p. 1-2].

In order to investigate how the transport capacity can be increased in the railway sector, a research group at SBB has developed a high performance physical simulation. Naturally, an increase in transport density, makes the transport network more complex, which makes an automated and computational approach a prospective solution. As such, the possibility of an automated Traffic Management System (TMS) is being explored. The TMS should handle all traffic on the railway network, which includes selecting train routes and the scheduling of trains. As a result, the TMS should also be able to address the VRSP. However, the complexity of the high performance physical simulator limits experimentation of new ideas and innovation. In order to address this issue, SBB, Société Nationale des Chemins de fer Français (SNCF), Deutsche Bahn (DB) and AICrowd, developed a simpler 2D-grid simulation environment called "Flatland". The Flatland environment's simpler nature allows for faster experimentation of novel ideas for a TMS, where promising approaches eventually can be implemented in the high performance physical simulation [Mohanty et al., 2020].

The Flatland project challenges people worldwide: Researchers, students, freelancers etc., to contribute with a solution to the VRSP railway problem. The challenge was first introduced in 2019 and has evolved along with iterations since then. In the first iteration, contributors mainly used solutions from Operations Research (OR). OR can be defined as prescribing mathematical analysis or models to solve issues in business practices. In this case, by simulating the environment of a railway network, to emulate the real world occurrence of the vehicle rescheduling problem (VRSP). The organizers have since the second iteration narrowed down their focus point and encourages their participants to implement Reinforcement Learning (RL) solutions, which are based on training an agent by rewarding or penalizing desired or undesired outcomes while it is interacting with an environment [Sutton and Barto, 2018, p. 1-2]. In the third and ongoing iteration in late 2021, an additional dimension has been added: timetabling. In the previous editions of the challenge, the trains were allowed to depart and arrive whenever. Timetabling has been added as it represents a fundamental part of real world railway-services [AICrowd et al., 2021b].

1.1 Flatland Challenge 2020 solutions

In order to get an overview of the ways in which a TMS in Flatland can be designed, the existing solutions are explored in the following section. One of the previous challenges is the Flatland challenge 2020, where the organizers released a paper of the most prominent solutions to the challenge [Laurent et al., 2021]. The solutions provided from the Flatland challenge in 2020 can in a broad sense be categorized into two groups, namely OR-based and RL-based solutions. Generally, the OR-based solutions outperformed the RL-based solutions. However, the RL-based solutions still displayed promising results [Laurent et al., 2021, p. 2]. To better understand why the OR-based solutions performed better, and what the underlying challenges are in the RL-based solutions, the next sections are devoted to a closer study of the two general approaches.

OR-based Solutions

Generally, the OR-based solutions were based on Multi-Agent Pathfinding (MAPF) algorithms combined with other optimization techniques [Laurent et al., 2021, p. 7]. The MAPF algorithms are well suited for the Flatland challenge, as the input to a classic MAPF algorithm is an unweighted graph, and a set of agents [Laurent et al., 2021, p. 7]. Essentially, the simulated railway network in the Flatland challenge can be represented as a graph with vertices and edges, where vertices represent possible transitions in a railway network and the edges present the paths between the vertices. The objective of the MAPF algorithms is then to calculate optimal and collision-free paths for all the agents [Laurent et al., 2021, p. 7]. Challenges with the MAPF-based approaches consists of effectively handling a large number of agents, which can increase the run-time rapidly. Also, they still need to handle rescheduling effectively simultaneously. Which led to a series of optimizations to be deployed by different teams.

Looking at the winning OR-based solution, they used Prioritized Planning (PP)[1] combined

[1]In short terms PP is an algorithm which assigns priority to agents, where the paths are planned in the order from high to low priority [Čáp et al., 2011, p. 837]

with Safe Interval Path Planning (SIIP)[2], to plan the shortest path for the agents while avoiding collisions [Laurent et al., 2021, p. 8]. To tackle the increased computational complexity, as a result of an increased number of agents, they introduced a lazy planning scheme. Here, the paths are initially only planned for some agents, which are then pushed to the environment. For the rest of the agents, their path is planned during execution [Laurent et al., 2021, p. 8]. This decreases the complexity as the path-finding algorithm does not have to plan the shortest and collision-free paths for e.g. thousands of agents at the same time. To tackle the randomized malfunctions which can result in deadlocks, they used Minimum Communication Policies (MCP), which abstractly maintains the ordering each agent visits a location by stopping some of the agents [Ma et al., 2011, p. 4].

This solution showcases the advantage of OR-based solutions, as they can effectively utilize the underlying grid-structure of the environment to plan collision-free paths for the agents. This leads to coordinated actions, which can mitigate the increased difficulty when multiple agents are dependent on each other's actions. However, even though the solution was the best performing in the Flatland challenge 2020, it still showcases the need for improvement when the scale of the environment and number of agents increases to large proportions [Laurent et al., 2021, p. 15].

RL-based Solutions

For the RL-based solutions, the general approach was to map the environment for an agent through a tree-based observation [Laurent et al., 2021, p. 8]. For instance, the railway network can be modelled by a tree-structure, where the nodes represent different switches, and the branches maps the paths between the switches to a defined tree depth in the environment. Different features can then be combined with the tree structure, e.g. presence of other agents on an agent's path [Laurent et al., 2021, p. 7]. Central to all RL-based solutions, is the use of a learning algorithm, where agents learn the appropriate actions through experience and interaction with the environment. Through iterations, each agent then learns what actions are favorable in order to reach its target destination, based on the rewards that are formulated to it. Different learning models were adopted, where the best RL-based solutions used Proximal Policy Optimization (PPO) (elaborated in section 4.1.1). However, the Flatland paper also concluded that the specific deployed training algorithm did not result in a very noticeable performance difference [Laurent et al., 2021, p. 16]. One of the key aspect and challenges with RL-based solutions in multi-agent environments are methods for improving coordination and communication between the agents to efficiently solve rescheduling and deadlocks.

The best performing RL-based solution, *PPO with communication and Departure schedule*, adopted the *Proximal Policy Optimization* (PPO) learning algorithm[Laurent et al., 2021, p. 10-11]. The input to the PPO model is features from a tree-based observation model, which provides the PPO model with information on the current state of the environment, such as the agent's current position. These features are furthermore used in a communication network, which collects information from all the agent, to for instance determine which agent has priority if two agents are located at a switch. This is to minimize

[2]SIIP is in broad terms an MAPF algorithm which plan collision free paths by assigning safe intervals between the agents [Mike phillips, 2011]

the probability of a deadlock occurring [Laurent et al., 2021, p. 10]. Another part of the solution is the use of a departure schedule, which limits the occupancy of the railway network. In short, the departure schedules makes use of a heuristic function to determine the order in which the agents enter the railway network. Specifically, the heuristic function calculates a predicted probability of an agent reaching its target.[Laurent et al., 2021, p. 11]. These adopted methods enhance the coordination between the agents and limits the probability of deadlocks occurring. However, the solution still scores lower than the best performing OR-based solution, especially when the size of the railway network and the number of agents grow.

1.2 Problem Scope

As stated earlier, the RL-based solutions are significantly outperformed by the OR-based solutions. However, the results of the Flatland challenge 2020 still display the potential of the RL-based solutions, as they to some extent overcome the difficult challenges of coordinating multiple agents, by adopting different communication and coordination methods. The Flatland paper also concludes that novel approaches are still needed to overcome the challenges of scalability, where existing RL-based solutions cannot effectively handle these. In regard to OR-based solutions, one of the key benefits is their ability to adopt long-term planning, where different planning methods can plan collision free paths, which mitigate the challenge of coordinating multiple agents. Nonetheless, the Flatland challenge 3 still encourages participants to develop solutions which leverage recent findings in RL [AICrowd et al., 2021b].

RL has been widely used in games, beating both bots and humans in the likes of GO[Silver et al., 2016] and Starcraft[Vinyals et al., 2019]. As such, Flatland, resembling the structure of a game, could potentially be susceptible to the utilization of similar methods. The promise of RL has been showcased in challenged previously pertaining to OR, e.g. RL outperforming classical heuristics in the Vehicle Routing Problem (VRP)[Nazari et al., 2018] and the traveling salesman problem (TSP)[Kool et al., 2019]. The Vehicle rescheduling problem (VRSP) has been heavily researched in OR, but persistent issues in regard to scaling to larger environment and number of agents[Gharibi et al., 2020], lays a reason for exploring the topic from a related field. As such, the novelty of RL, its potential and fundamental principle of learning from interaction and experience, which closer resembles how humans and animals learn, makes it an interesting topic of research. For this reason, this project investigates the prospect of RL, and how it can be used to develop a TMS within the Flatland environment. In order for a TMS to be considered applicable to a real world setting, it should have the following qualities:

- **Generalization:** Railway networks are not identical, and infrastructure and schedules are unavoidably subject to change. As such, the TMS should have the ability to generalize to different environments and schedules.

- **Scalable:** Railway traffic can be dense and complex. For instance, in Switzerland more than 10,000 trains run each day, and the railway infrastructure consists of over 13,000 railway switches and 32,000 signals [AICrowd et al., 2021a]. Consequently, the TMS should be scalable to railway traffic on more complex configurations of railway networks.

- **Adaptable:** With a complex environment, unforeseen circumstances are inevitably bound to happen, such as a train malfunctioning. Even a small disturbance, as this, can have a significant impact on service quality and stability of the entire railway network [Laurent et al., 2021, p. 2]. As a result, the TMS should be able to adapt to unpredictable events and handle such instances accordingly by dynamic rescheduling.

Problem Statement

In light of the above, the following problem statement is derived:

How can RL be used to develop a TMS for the Flatland environment, which is capable of generalization, scalable to complex railway networks, and is adaptable to unpredictable events.

Firstly, the Flatland environment is explored in chapter 2. Subsequently, chapter 3 provides an overview of the relevant RL theory. Afterwards, chapter 4, describes the design and development of the RL-solution as well as formulating the input from the environment, in an optimized format and methods to further enhance performance. Following this, the developed RL-solution's performance – in regard to the three formerly mentioned qualities – is evaluated in chapter 5. Finally, the results of the evaluation are then discussed and the developed RL-solution is concluded upon in chapter 6 and 7, respectively.

The Flatland Environment 2

This chapter will introduce the Flatland environment in depth, along with its essential aspects, which will be used to develop and test a TMS solution. The environment involves different facets such as cells, agent actions, observation models. Flatland has a graphical interface where the behavior of the agents and the generated world can be observed, an example of which can be seen in Figure 2.1.

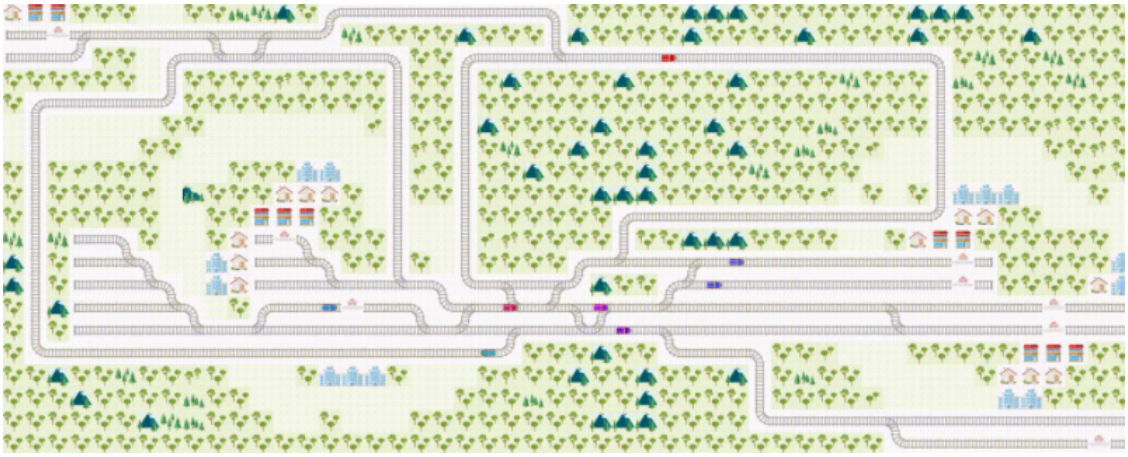


Figure 2.1: Example of the graphical interface of the Flatland environment

2.1 Land and Tracks

Flatland is a 2D grid environment consisting of cells, and can be of an arbitrary size. To model a real world railway network, each cell can have eight different states as shown in Figure 2.2.

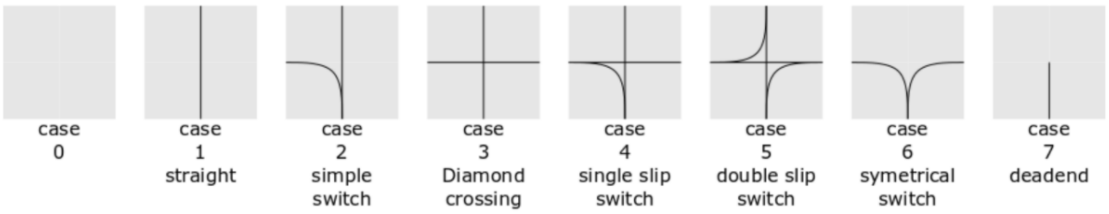


Figure 2.2: Visualization of the 8 different cells in the Flatland environment [AIcrowd, 2021c]

While an agent is on the grid, it has a specific x,y coordinate on the grid as well as a directional integer value between zero and three. The position of an agent will always be

unique, as each cell can contain at most a single agent at a time. The four directional values are a representation of the four cardinal directions North, South, East, West. The directional values have an effect on how the agent makes a decision on the grid. For example, if an agent approaches a double slip switch from the South, it can either choose to continue straight (North) or go right (West). On the other hand, if it approaches the simple switch from the North, it can either choose to continue straight (South) or go right (West). This results in the double slip switch accepting different actions depending on the direction of the agent, and the same logic applies to the other cell configurations as well.

Agent Actions

The different cells have, as mentioned above, an effect on which actions the agent can take given a location and direction in the environment. Flatland has a predefined action space which defines five allowed actions for the agents:

- **Do nothing:** The agent continues doing what it was previously doing. E.g. if an agent is already moving forward, and do nothing is the selected action, then it will continue to do so. If an agent does an illegal action, such as turn into another field that is already occupied, that action will be cancelled, and it will revert back to this action instead.

- **Move left:** The agent will move to the adjacent cell to the left from its current direction. Thus, a left turn can only be executed when an appropriate switch is present. If the move left action is chosen, but it is not a valid transition on the given cell, then the action move forward will be executed instead.

- **Move right:** This action is in principle the same as the "Move left" action, except that the agent will make a right turn.

- **Move forward**: The agent will move to the next cell in its current direction, if it is a valid action. This action will start a forward motion until another action is executed, or an obstruction makes the action impossible.

- **Stop moving**: This action stops the agent.

Agents can pick any action they see fit. However, in order to simulate malfunctions in the Flatland environment, an agent can randomly malfunction. When an agent malfunctions, it will stop and block the track for other agents for a random amount of time within a given duration interval [AIcrowd, 2021c]. The malfunctions of Flatland is a main component of what makes it a VRSP, as described in section 1, as this is a problem that arises when a previously assigned trip is disrupted.

2.2 Observations in Flatland

There are three different pre-configured implementations of the observation space provided in Flatland. These are shown in Figure 2.3. An observation serves as a description of the environment at a given time for an agent.

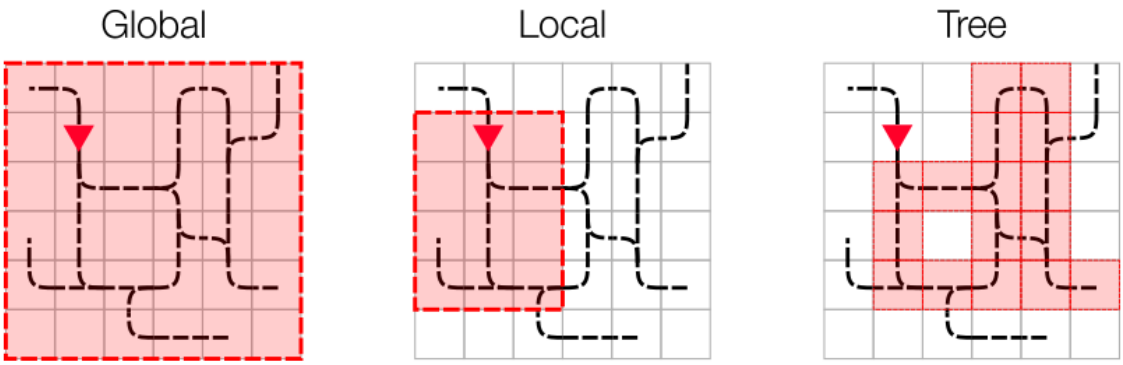


Figure 2.3: Visualization of the three provided observation models by Flatland: global, local and tree observation [AIcrowd, 2021c]

- **Global:** This observation provides information of the entire environment. This information is divided into five different channels: Representation of the agent's own position and direction, the positions and directions of other agents, all malfunctions as soon as they occur, all speed variations of agents, and the number of agents ready to depart from their station.

- **Local:** Local observation is when an agent's information of the environment is bound to a limited number of cells around it. This observation has the five same information channels as the global observation.

- **Tree:** This observation represents the railway network as a graph structure. An illustration of the tree observation method can be seen in Figure 2.4. The tree structure is a quad tree where each branch corresponds to the actions of the agent i.e. forwards, left, backwards and right. However, the tree only builds a branch upon nodes which contains a valid transition. This can be seen in Figure 2.4 in the transition from level zero to level one. Here, only one branch is built, as the only valid transition the agent can make is a forward action. The same principle applies when the depth goes from level one to level two, where new branches are built from the forward and right actions. This tree structure then keeps building until it reaches a defined max depth.

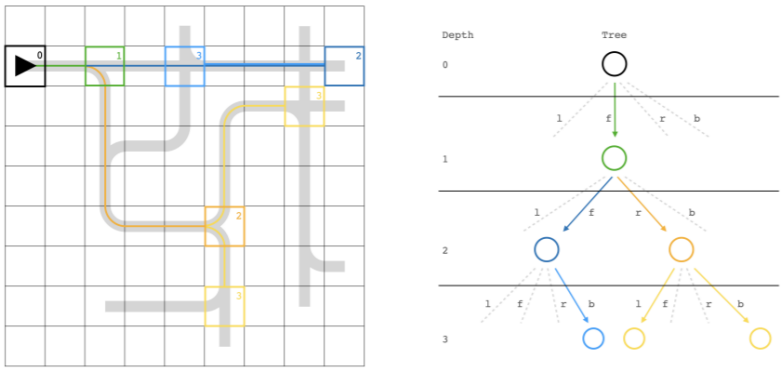


Figure 2.4: Illustration of the Flatland tree observation. On the left picture is an illustration of the nodes and branches which can be observed from a depth of three. On the right picture it is showcased that a branch can only be expanded from the actions corresponding to the allowed transitions in the environment

Speed Profiles

Different types of trains travel at different speeds. For instance, a freight train usually moves slower than a passenger train. Different speed profiles naturally introduce further complexity to scheduling. The Flatland environment has five speed profiles for the agents per environment, which are randomly assigned values between zero and one, where one is the fastest possible speed and zero the slowest. In practice, this means that a train with a speed of one, will move one cell each time step, whereas a train with a speed of a quarter will use four time steps to do the same [AIcrowd, 2021e].

Timetable

Timing and punctuality is of essence for real-world railway traffic. As a result, agents have scheduled departure and arrival times. Timetabling is handled by giving an agent two attributes: the earliest time step at which it can depart, as well as the latest time step it is expected to reach its destination at [AIcrowd, 2021g]. This introduces complexity as previous iterations of the Flatland challenge the agents simply had to arrive at their target. Now they have to arrive on time as well, or they will get an increasing penalty corresponding to the amount of time they arrive later than their expected arrival time.

Rewards

Flatland runs on a sparse reward setting which is evaluated when an episode ends, either when all trains have arrived at their destination, or the maximum amount of time steps has been reached. There are three clauses which affect the reward:

- *The agent has arrived at its destination:* In this scenario, the agent is given a reward of 0 for either arriving on time or before time.

- *The agent did not reach its destination in the allocated amount of time steps:* A penalty is given, which is based on an estimate of how many time steps it will take the agent to reach its destination from its current position.

- *The train never departed:* The train is considered to be cancelled and a penalty is given based on the travel time for the shortest path between origin and destination [AIcrowd, 2021c].

In summation, the best possible reward is therefore a value of 0, which is achieved by reaching the target on time. The penalties of not reaching the target is based on the required travel time to reach the target.

Reinforcement Learning 3

This chapter introduces the preliminaries needed to design and develop a RL-based TMS for Flatland. As such, this chapter explores the central ideas of RL and a related field, supervised learning. The RL-theory in this chapter has Sutton and Barto [2018] as its primary source of inspiration.

Machine Learning (ML) conceptualizes the idea of not having to explicitly program the computer's decisions. ML algorithms use data to train a model to make decisions or predictions [Sutton and Barto, 2018, p. 1-5]. The field of ML can in general be divided into three categories:

- **Supervised Learning**: With Supervised Learning, a mathematical model is trained through a data set with desired outputs (labeled data). The task is to approximate a desired function. This is done through training, where it is given examples of the desired input-output behavior, which it learns to mimic. Supervised learning algorithms are most commonly used for regression and classification problems [Sutton and Barto, 2018, p. 2].

- **Unsupervised Learning**: With Unsupervised Learning, the task is to self-discover patterns occurring in an unlabeled training data set. As such, unsupervised learning algorithms are most commonly used for clustering and association rule learning problems [Sutton and Barto, 2018, p. 2].

- **Reinforcement Learning**: Reinforcement learning is a computational approach of independently learning what to do from interaction and experience, with the goal of maximizing a numerical reward signal. Reinforcement Learning algorithms are most commonly be used for OR, game theory, control theory and more [Sutton and Barto, 2018, p. 2-3].

In order to successfully communicate machine learning practices, a mathematical formalization of the domain can benefit in the construction of the particular model that is being designed.

3.1 Markov Decision Processes

Markov Decision Processes (MDPs) is a mathematical framework of learning through interaction with a goal in mind [Sutton and Barto, 2018,p. 47]. Specifically, they conceptualize the sequential decision-making for an agent, which interacts with an *environment* – everything outside the agent. This agent-environment interaction is visualized in Figure 3.1.

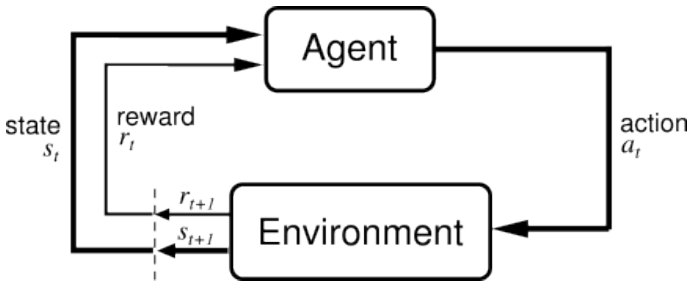


Figure 3.1: Depiction of the agent-environment interaction of an MDP. Whenever an agent takes an action a_t in a given state s_t, the environment responds by presenting a new state s_{t+1} to the agent. Each action taken by the agent is rewarded either positively or negatively based on a numerical reward, r_{t+1}. The agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, ...n$ [Sutton and Barto, 2018,p . 48].

The preliminaries needed to understand the fundamental methods within the RL paradigm are as follows:

- **Agent**: An agent is a learner and decision maker in an environment.
- **State**: A state s_t is a description of the environment at a given time step t.
- **Observation**: An observation is a partial or fully description of the environment, restricted by e.g. the agent's sight [Sutton and Barto, 2018, p. 464]
- **Environment**: An environment is the space wherein the Agent takes actions.
- **Action**: An action a_t is taken at time step t by the agent to transition to a new state s_{t+1}.
- **Reward**: A reward $R_a(s, s')$ is the positive or negative feedback the agent receives from the environment based on its action.
- **Policy**: A policy π defines the agent's decision-making at a given time. [Sutton and Barto, 2018, p. 1]
- **Trajectory**: A trajectory τ is the sequence of states, actions and rewards an agent encounters over time: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$ [Sutton and Barto, 2018, p. 48].

In a finite MDP, the sets of states $\mathbb{S}$, actions $\mathbb{A}$ and rewards $\mathbb{R}$ have a finite number of elements. The random variables R_t and S_t have well-defined discrete probability distributions only dependent on the previous state and action [Sutton and Barto, 2018, p.48-49]. In other words, the probability of particular values of these random variables $s' \in \mathbb{S}$ and $r \in \mathbb{R}$ for occurring at time t, are dependent on values of the previous state and action. As such, it is formalized [Sutton and Barto, 2018, p.48-49]:

$$p(s', r|s, a) = Pr[S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a], \tag{3.1.1}$$

for all $s', s \in \mathbb{S}$, $r \in \mathbb{R}$, and $a \in \mathbb{A}(s)$. In an MDP, the probabilities given by function p defines the environment dynamics. That is, the probability of each possible value for S_t and R_t depends only on the immediate previous state S_{t-1} and action A_{t-1} [Sutton and Barto, 2018, p.48-49].

Every MDPs' states should satisfy the *Markov Property*. A state that is capable of retaining all relevant information from the previous agent-environment interactions into the present to predict the future, has the Markov property [Sutton and Barto, 2018, p. 49].

Though later in Section 3.2, approximation methods that do not rely on the Markov Property are introduced.

Goal, Discounted Return and Episodes

The goal of an agent is to maximize the *expected return*. That is, the expected value of the cumulative sum of rewards. As such, the goal is not to maximize the immediate rewards, but instead the long-term cumulative rewards. Task with episodes – agent-environment interaction breaks into sub-sequences – are called *episodic tasks* as opposed to agent-environment interactions that does not have terminal states, which are called *continuous tasks*. For episodic tasks, the expected return G can, in its fundamental form, be defined as the sum of the rewards. The agent's objective is to maximize the expected return, which is what guides the agent to make its decisions. In its simplest form, it can be defined as [Sutton and Barto, 2018,p. 53-55]:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{3.1.2}$$

where T is the final time step. Each episode ends in a terminal state for the agent, which is followed by a reset to a default starting state or to a sample of a Gaussian distribution of starting states. The next episode begins independently of the previous one. In general, episodes end in the same terminal state with different rewards for the different outcomes. When taking continuous tasks into consideration, the expected return from Equation (3.1.2) is problematic, since the final time step T would be $T = \infty$, which means that the expected return could in turn be infinite [Sutton and Barto, 2018,p. 53-55]. In light of this problem, a *discount factor* can be introduced. This factor controls the present value of future rewards. In other words, how much agents care about the immediate future in relative to the distant future. The expression of the total reward for both continuous and finite *horizons* – the amount of time steps an agent looks forward from time step t – taking the discount factor into account [Sutton and Barto, 2018,p. 55]:

$$G_t = R_{t+1} + \gamma R_{t+2}\gamma^2 R_{t+3} + ... = G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{3.1.3}$$

where the discount rate γ is a hyperparameter, $0 \leq \gamma \leq 1$. The infinite sum of Equation (3.1.3) has a finite value, if $\gamma < 1$ and the sequence of rewards is bounded. The closer γ is to zero, the more the agent becomes *myopic*. A myopic agent would act to only maximize

each immediate reward, which would possibly reduce access to future results in a reduced return. The closer γ is to 1, the more the agent becomes *far-sighted*, where it will evaluate each of its actions based on the sum of all of its future rewards [Sutton and Barto, 2018, p. 55].

3.1.1 Policy and Value Functions

Value functions of states estimates how good or bad it is to be in a given state or taking a particular action in a given state. This is calculated based on the expected return. Naturally, the agent's future rewards depends on the action it will take. This is where the definition of an agent's way of behaving will be defined: *policy*. A policy π is a mapping from perceived states to probabilities of selecting each action. A policy can be either *deterministic* or *stochastic*. A deterministic policy has a deterministic action for each state. A stochastic policy has an action probability distribution for each particular state.

The value function of a state s under a policy π, denoted $v^\pi(s)$, is the expected return [Sutton and Barto, 2018, p. 58]:

$$v^\pi(s) = \mathbb{E}_\pi \left\{ G_t \mid s_t = s \right\} = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid S_t = s \right] \tag{3.1.4}$$

for all $s \in S$ and t being any arbitrary time step. $\mathbb{E}_\pi$ denotes the *expected value* of a *random variable* following a policy π. The function v^π is called the *state-value function for policy* π. To define the value of taking a particular action in a state, is very similar to state-value function. Here, a given action a is also taking into consideration:

$$q^\pi(s, a) = \mathbb{E}_\pi \left\{ G_t \mid s_t = s, A_t = a \right\} = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid S_t = s, A_t = a \right] \tag{3.1.5}$$

for all $s \in \mathbb{S}$ and $a \in \mathbb{A}$. q^π from Equation (3.1.5) is called the action-value function for policy π [Sutton and Barto, 2018, p. 58]. These value functions v^π and q^π can be estimated solely from the agent's experience, methods to do this are expanded upon in Section 3.2.2.

Optimal Policy and Value Functions

Finding a policy that maximizes the expected return, roughly solves an RL task. An optimal policy can be defined for finite MDP's. Value functions define an ordering of policies; a policy π is defined to be equal to or better than a policy $v^{\pi'}$ if the policy π expected return is greater than or equal to policy π' for all states. Formally, $\pi \geq \pi'$ if and only if $v^\pi(s) \geq v^{\pi'}(s)$ for all $s \in S$. Policies that are equal to or better than all other policies is called an *optimal policy*. There is always at least one policy that is an optimal policy. All optimal policies are denoted as πv^*. The optimal policies all share the same state-value function, denoted $v^*(s)$. They are called *optimal state-value function* and are defined as following [Sutton and Barto, 2018, p. 62-63]:

$$v^*(s) = \max_\pi v^\pi(s) \tag{3.1.6}$$

for all $s \in \mathbb{S}$. Similarly, the optimal policies share the same *optimal action-value* function [Sutton and Barto, 2018, p. 62-63]:

$$q^*(s,a) = \max_\pi q^\pi(s,a) \tag{3.1.7}$$

for all $s \in \mathbb{S}$ and $a \in \mathbb{A}(s)$. The function gives the expected return, for each state-action pair (s,a), for taking action a in state s and subsequently following an optimal policy. q^* can then be defined as follows [Sutton and Barto, 2018, p. 63]:

$$q^*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v^*(S_{t+1}) \mid S_t = s, A_t = a\right] \tag{3.1.8}$$

Exploration Versus Exploitation

One of the most fundamental problems within reinforcement learning algorithms is the *exploration vs exploitation trade-off dilemma*. Recall that, the agent learns and maps which actions to take to maximize the reward signal. The agent's decision-making is not explicitly coded, but must discover which action yields the most reward. This is done through trial-and-error. [Sutton and Barto, 2018, p. 25]

The notion of decision-making under uncertainty in RL is formalized in the *Multi-armed bandit problem* [Sutton and Barto, 2018, p. 25-27]. In this problem, the agent chooses between k-actions and receives a reward based on this action. Assume that each action has a distribution of rewards corresponding to the notion of desirable or undesirable action. At least one of these actions generates a maximum numerical reward. Thus, each action's reward probability distribution is different and is unknown to the agent. The goal of the agent is to identify which action to select to get the maximum reward through trial and error. [Sutton and Barto, 2018, p. 25-27].

- *Exploration*: To expand the agent's knowledge about each action, it must explore actions that are not well-explored. By exploring less-known actions, it enables the agent to improve the accuracy of the estimated action values, and therefore the agent will be able to make more informed decisions in the future.

- *Exploitation*: Here, the agent exploits its current knowledge, utilizing its current action-value estimates of most reward and selects a greedy action. The *greedy* actions is the actions that have the highest estimated value at a given time step. This sounds good in theory, however it may not get the most reward in the long run and will ultimately lead to suboptimal behavior. Reason being, that the agent has not yet discovered the most accurate estimates of action-values on the non-greedy actions [Sutton and Barto, 2018, p. 26-27].

An agent gets more accurate estimates of action-values when exploring, and will get more reward by exploiting. It cannot, however, do both simultaneously. This is called the *Exploration versus Exploitation*-dilemma. In the short run during exploration, the reward

may be lower, but after exploring the reward will be higher, discovering better actions; thus the agent will be able to exploit them many times.

To get the most out of both worlds, by balancing exploration and exploitation, a basic method called ϵ-greedy, where $0 < \epsilon < 1$ controls the amount of exploration vs exploitation. It is used to choose between exploration and exploitation until full knowledge about the action-value estimates of each action is achieved. However, in any RL-case, the Exploration versus Exploitation-dilemma is complex, as it depends on precise values of the estimates, uncertainties, and the number of remaining steps. There are many sophisticated methods to balance exploration and exploitation in the multi armed bandit problem, for example the ϵ-greedy method, however most of these methods makes strong assumptions about stationarity and prior knowledge that are either impossible or violated in applications [Sutton and Barto, 2018, p. 26-27].

Approaches to Finding an Optimal Policy

Naturally, after having the optimal policy abstractly defined using Equation (3.1.8), the question arises: how to find the optimal policy. Recall that the agent follows a stochastic policy when taking actions based on a probability distribution in given states. In order to find an optimal policy, there are two categorical approaches: *off-policy* methods and *on-policy* methods. This is visualized in Figure 3.2.

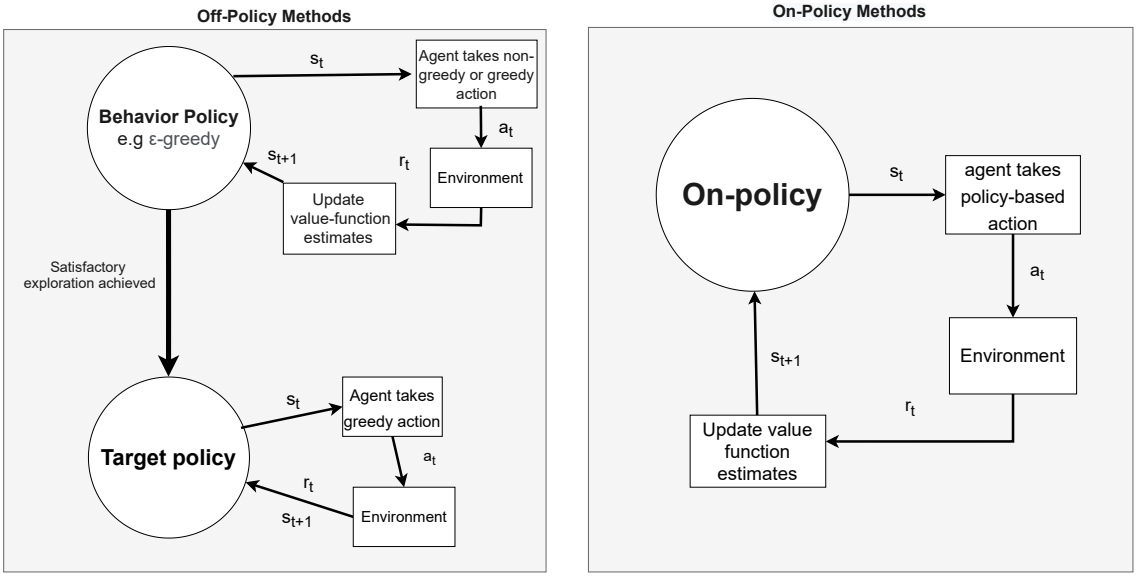


Figure 3.2: When using an on-policy method, the policy is evaluated sequentially as the agent is taking decisions. Off-policy methods uses a policy to evaluate, which differs from the one used for generating the data [Sutton and Barto, 2018, p. 100].

The appropriate method from the two seen in Figure 3.2 is utilized depending on the use case. Generally, the two approaches attempts to balance exploring and exploitation which was introduced in Section 3.1.1. As seen in Figure 3.2, off-policy methods distinguishes between a *target policy* and a *behavior policy*, where the behavior policy is the one generating the behavior and the target policy is the one being learned about. A simple method to employ in the behavior policy could be ϵ-greedy to accurately approximate the

value functions by exploring. When the value functions are estimated accurately enough, the learned target policy will be employed to take the best possible actions. An on-policy uses the same policy for behaving and learning concurrently. In other words, at each time step it will take an action and update its value functions simultaneously to find the optimal policy [Sutton and Barto, 2018, p. 100-104].

3.2 Approximate Methods

In most cases, the state space of an environment will be immense, and finding the optimal policy or value function would be an infeasible pursuit. Thus, the only way to solve this problem is to find an approximate solution. The main problem that arises with vast state spaces, is that every state encountered will by and large never have been observed before. In order to make sensible decisions in an environment where most states are foreign, it is essential for the RL-model to have the ability to generalize from previously encountered similar experiences. Thus, what is sought after is an RL-model competent in *generalization*. Fortunately, supervised learning concerns itself with this, and its methods can be used to find an approximation of the optimal policy or value function [Sutton and Barto, 2018, p. 195].

Within the domain of supervised learning, non-linear function approximation, in particular MLPs (Multilayer Perceptrons), have played a key role in forming the basis of deep learning[1] methods and state-of-the-art reinforcement learning algorithms [Sutton and Barto, 2018, p. 223]. As such, MLPs are of particular interest.

[1] Neural networks with more than one hidden layer, and usually a more complex network architecture.

3.2.1 Multilayer Perceptron

An MLP have loosely been inspired by the findings of neuroscience. Roughly speaking, it can be viewed as a superficial and largely simplified mathematical model of the biological brain [Goodfellow et al., 2016, p. 169]. An MLP is a network of connected units, called *neurons*. In its essence, an MLP can be viewed as a directed acyclic graph arranged in layers, where information is only passed forward from one immediate layer to another. The architecture of an MLP is visualized in Figure 3.3.

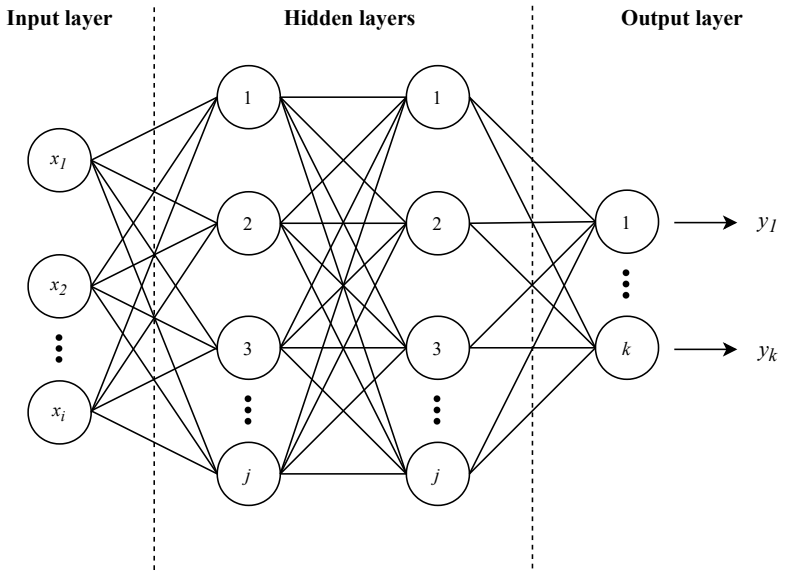


Figure 3.3: The architecture of an MLP with a *forward pass* from left to right. Each neuron has an activation, a, and each connection between two neurons has an associated weight, w, to define the connection's strength. The first layer is called the *input layer* as it takes in an input vector, $\boldsymbol{x}$. The last layer is called the *output layer* as it produces a output vector, $\boldsymbol{y}$. The intermediary layers are called *hidden layers*, and can consist of one or more layers. The figure is from one of the authors' previous works [Bonvang et al., 2020, p. 34].

Hidden Neurons

Each hidden neuron acts as a vector-to-scalar function, where each hidden neuron takes as input the weighted sum of activation values of all the neurons from the preceding layer along with a bias, b. E.g. for a hidden neuron, n_j, in the earliest hidden layer of the MLP shown in Figure 3.3, the input is [Russell and Norvig, 2010,p. 727]:

$$in_j = \sum_{i=1}^{J} w_{i,j} \cdot a_i + b_j \tag{3.2.1}$$

Thereafter, in order to make the hidden neuron's activation relatively binary, in which it is either active or inactive, a differentiable non-linear *activation function*, h, is applied to Equation (3.2.1). As a result, the activation of a hidden neuron is expressed as [Russell and Norvig, 2010,p. 727]:

$$a_j = h(in_j) \tag{3.2.2}$$

The introduction of nonlinearity expands the MLP's modelling capacity, and makes it possible for MLPs to recognise patterns beyond linear relationships [Goodfellow et al., 2016, p. 169]. There are many types of activation function available, of which the three most commonly used are shown in Figure 3.4.

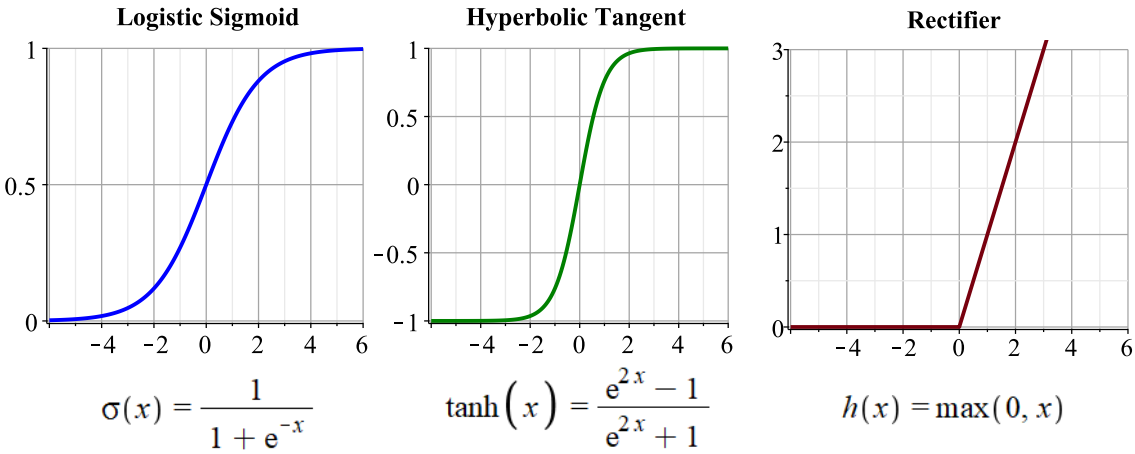


Figure 3.4: The three most commonly used activation functions for hidden neurons. The figure is from one of the authors' previous works [Fagerlund et al., 2021, p. 10].

Naturally, the question which arises from this, is what activation function to use. In this regard, one should keep in mind that models are easier to train the closer they resemble the behavior of a linear model. In light of this, a desirable trait of non-linear activation function is that it has linear characteristics [Goodfellow et al., 2016, p. 194].

First and foremost, as seen in Figure 3.4, the two sigmoidal functions (logistic sigmoid and hyperbolic tangent) bear close semblance to one another. However, the hyperbolic tangent (Tanh) is usually favoured over logistic sigmoid. Around $x = 0$ Tanh is more akin to the identity function, which results in a model behavior closer to linear, making it easier to train. Nonetheless, a downside of using either sigmoidal functions is that both inherently *saturate*. Both functions map any arbitrary input to a bounded interval. E.g. for a logistic sigmoid, negative and considerably small values are mapped to 0, and likewise considerably large values are mapped to 1, making it only particularly sensitive to its input near 0. As a result, the two sigmoidal functions have a limited sensitivity to their input due to saturation, which makes the model harder to train. The Retificer Linear Unit (ReLU), on the other hand, does not saturate and more closely resembles a linear function, as it consists of two linear pieces. As such, the ReLU is usually regarded as the preferable among the three [Goodfellow et al., 2016, p. 193 - 195].

That being said, the above should only provide some intuition, and intuition alone is not sufficient, as there in a general sense is no particular ideal activation function. It greatly varies from task to task, and finding the ideal activation function is inescapably a matter of trial-and-error [Goodfellow et al., 2016, p. 192].

Training Goal

From Figure 3.3, an MLP can be viewed as a series of functional transformations, which defines a deterministic mapping from $\boldsymbol{x}$ to $\boldsymbol{y}$ [Goodfellow et al., 2016, p. 183]:

$$\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta}) \tag{3.2.3}$$

As such, the goal of training an MLP is for it to learn the values of the parameter (weights and biases), $\boldsymbol{\theta}$, that produces the best function approximation. Exactly how the values of $\boldsymbol{\theta}$ are adjusted depends on the *objective function*, which is aimed to be either minimized or maximized. Maximization is achieved by minimizing $-f(\boldsymbol{x}; \boldsymbol{\theta})$.

In supervised learning, the goal is generally to minimize the objective function, where the principles of *maximum likelihood estimation* are applied. Maximum Likelihood Estimation (MLE) is an estimate of $\boldsymbol{\theta}$ where the empirical data is most likely to occur [Pishro-Nik, 2014, section 8.2.3]. Let $\mathbb{X} = \{x_1, ..., x_m\}$ be a sample drawn i.i.d (independent and identically distributed) from the true data distribution, $p_{\text{data}}(\boldsymbol{x})$, and let $p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta})$ be the model distribution which estimates $p_{\text{data}}(\boldsymbol{x})$. MLE is then defined as [Goodfellow et al., 2016, p. 131-132]:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \; p_{\text{model}}(\mathbb{X}; \boldsymbol{\theta}) \tag{3.2.4}$$

$$= \arg\max_{\boldsymbol{\theta}} \; \prod_{i=1}^{m} p_{\text{model}}(\boldsymbol{x}_i; \boldsymbol{\theta}) \tag{3.2.5}$$

$$= \arg\max_{\boldsymbol{\theta}} \; \sum_{i=1}^{m} \log p_{\text{model}}(\boldsymbol{x}_i; \boldsymbol{\theta}) \tag{3.2.6}$$

$$= \arg\max_{\boldsymbol{\theta}} \; \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}} [\log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta})] \tag{3.2.7}$$

The product in Equation (3.2.5) can be prone to numerical underflow – a rounding error where numbers near zero are rounded to zero. For this reason, a logarithmic probability, Equation (3.2.6), is much more practical, as it conveniently transforms it into a sum instead, which also makes it faster to compute. Finally, MLE can be expressed as an expectation (by dividing with m) as shown in Equation (3.2.7) [Goodfellow et al., 2016, p. 132]. Most commonly for supervised learning, the goal is to estimate a conditional probability, where $\boldsymbol{y}$ has to be predicted given $\boldsymbol{x}$. The conditional MLE is expressed as [Goodfellow et al., 2016, p. 133]:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \; \mathbb{E}_{\boldsymbol{x}, \boldsymbol{y} \sim p_{\text{data}}} [\log p_{\text{model}}(\boldsymbol{y} | \boldsymbol{x}; \boldsymbol{\theta})] \tag{3.2.8}$$

With the principles of MLE, the goal of training an MLP can be viewed as an attempt to make the model distribution match the empirical distribution (defined by the training set). As a result, there is a need for quantifying the loss between them, in order to minimise their dissimilarities. In this regard, *cross entropy*, H, can be used as a measurement [Goodfellow et al., 2016, p. 179]:

$$H(p_{\text{data}}, p_{\text{model}}) = -\mathbb{E}_{\boldsymbol{x}, \boldsymbol{y} \sim p_{\text{data}}} [\log p_{\text{model}}(\boldsymbol{y} | \boldsymbol{x}; \boldsymbol{\theta})] \tag{3.2.9}$$

In light of this, finding the MLE is equivalent to minimizing the cross entropy. As such, in supervised learning the objective function, in a general sense, is the cross entropy between the two probability distributions, and the goal is to minimise it in order for the MLP to learn the appropriate values of the parameter $\boldsymbol{\theta}$.

Output Neurons

The particular form of the objective function depends on how the output should be represented, and the choice of how to represent the output depends on problem that the MLP is tasked to solve. An MLP can be tasked to solve many different problems, of which the two most common are:

- Regression: Predict one or more numerical values from an input vector $\boldsymbol{x}$ [Bishop, 2006, p. 137].
- Classification: Assign an input vector $\boldsymbol{x}$ to one of K discrete classes [Bishop, 2006, p. 179].

Accordingly, the output layer has the purpose of providing a final transformation, such that the model's output coheres with the intended task [Goodfellow et al., 2016, p. 181].

If it is a regression problem, a linear activation function is usually used in the output layer. Here, no transformation is applied, and each output neuron's activation is simply their weighted sum. As the model predicts a real-value, the model can be considered to predict the mean of a Gaussian distribution. As such, the objective function is the cross entropy between the data distribution and a Gaussian distribution. In particular, the objective function J can be expressed as the Mean Squared Error (MSE) [Goodfellow et al., 2016, p. 132-134]:

$$J^{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - t_i)^2 \tag{3.2.10}$$

Where:

y_i	The predicted value
t_i	The observed target value
N	The number of training examples

On the other hand, if it is a classification problem, the MLP is tasked to output either a Bernoulli distribution (two classes) or Multinoulli distribution (more than two classes). In both cases, the output layer has a single output neuron. For a binary classification problem the logistic sigmoid activation function is used, as a Bernoulli distribution is defined by a single probability. The objective function is, as a result, the cross entropy between the data distribution and a Bernoulli distribution, called the *binary cross entropy*. For more than two classes, the softmax activation function is used represent a probability distribution over n classes. The softmax function produces a vector, where each element is between 0 to 1, and the entire vector sums to 1 [Goodfellow et al., 2016, p. 182, 184]. The objective function is, consequently, the cross entropy between the data distribution and a Multinoulli distribution, called the *categorical cross entropy*.

Learning

Initially, the parameter $\boldsymbol{\theta}$ is set randomly, after which the values of $\boldsymbol{\theta}$ are adjusted iteratively. This is usually done by the use of either *gradient descent* or *gradient ascent* – both of which are iterative optimization algorithms. Gradient descent aims to minimize the objective function. As shown in Equation (3.2.10), this is achieved by minimizing the loss between the current output and its target value. In brief, gradient descent finds a local minimum by computing the gradient of the objective function with respect to $\boldsymbol{\theta}$. For each iteration the values in $\boldsymbol{\theta}$ are then adjusted with a step of size α (also known as the learning rate) in the direction opposite of the gradient – the direction which minimises the objective function. In other words, a step is taking 'downhill' for each iteration. Conversely, gradient ascent does the exact opposite. Instead, a step is taken 'uphill' for each iteration, in order to find a local maximum [Goodfellow et al., 2016, p. 82-86].

As previously mentioned, an MLP can be viewed as a series of functional transformations. For instance, a four-layered MLP, as shown in Figure 3.3, can be presented as follows [Goodfellow et al., 2016, p. 168]:

$$f(\boldsymbol{x};\boldsymbol{\theta}) = f_4(f_3(f_2(f_1(\boldsymbol{x})))) \tag{3.2.11}$$

with f_1 as the input layer, f_2 as the subsequent layer, and so forth. As such, the layers are chained together and the output layer, f_4, depends on all the layers prior to it. Deducing how f_4 depends on f_3 can be done by calculating the gradient of the objective function. After which, computing how f_4 depends on f_2 is done by the use of the *chain rule* of calculus. Subsequently, computing how f_4 depends on f_1 can be done by applying the chain rule again. Summarily, deducing how f_4 depends on all the layers prior to it, is a matter of recursively applying the chain rule. For this reason, the method for computing the gradients for the whole MLP is known as *backpropagation*, as the gradients are computed in a *backward pass*, from back to front [Goodfellow et al., 2016, p. 205-208].

In order for an MLP to become sufficiently capable in generalization it has to be trained with a relatively large training set. As a result, gradient methods are usually quite computationally expensive and time-consuming, as a *training pass* consists of a forward pass with the whole training set, followed by a backward pass. As such, gradient methods have by and large been replaced with more computationally efficient sampling-based versions known as *stochastic gradient methods*. Here, a gradient is considered an expectation, which has statistically been estimated using a set of samples, known as a *minibatch*. In order to obtain an unbiased estimate of the gradient, each sample has to be drawn i.i.d. In practice, this is usually achieved by shuffling the training set once for each epoch – a full pass through the entire training set. Summarily, with *stochastic gradient methods* the computation time for a training pass is no longer bound to the size of the training set, but instead depends on the size of the minibatch [Goodfellow et al., 2016, p. 151-153, 280].

However, the random sampling inevitable introduces noise. As the gradient is estimated from a set of randomly picked samples, it will no longer converge directly towards a local optimum. Instead, it will converge in an oscillating manner, as shown in Figure 3.5.

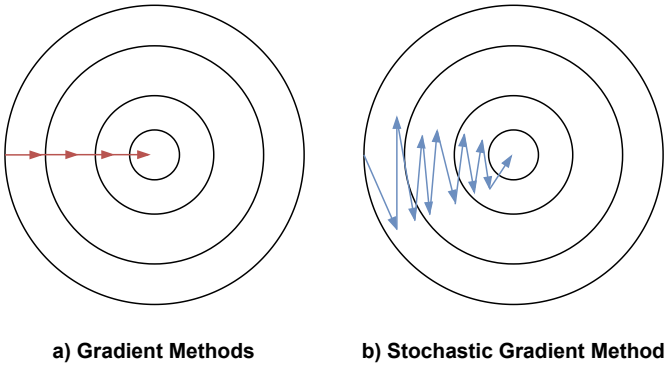


Figure 3.5: Conceptualization of the convergence behavior of gradient methods compared to stochastic gradient methods. The figure is modified from one of the authors' previous works [Fagerlund et al., 2021, p. 39].

Consequently, the computational speed gained with stochastic gradient methods is at the expense of a slower convergence rate. Several methods exist to mitigate this issue [Goodfellow et al., 2016, p. 294]. Currently, one of the predominant methods is the *Adam optimizer*, as it is resource efficient – both computationally and memory-wise – and compares favorably to its counterparts. Furthermore, the Adam optimizer is also appropriate for non-stationary objectives, which makes it favorable in the context of RL, as RL concerns itself with non-stationary objective functions [Kingma and Ba, 2014] [Sutton and Barto, 2018, p. 195]. In order to mitigate the noise introduced by stochastic methods, Adam implements *momentum*, which, in short, is a method to accelerate learning when the convergence rate is impeded by noise. Furthermore, the learning rate has a substantial influence on the MLP's performance. A smaller learning rate will result in more precise weight adjustments, but at the cost of slower convergence. However, if it is too small the model is at risk of getting stuck at a sub-optimal local optimum. In contrast, a larger learning rate will produce a faster convergence, but at the cost of accuracy. If it is too large, on the other hand, the model might never converge to any optimum [Goodfellow et al., 2016, p. 429]. Adam addresses this issue by computing individual adaptive learning rates, meaning each weight in the MLP has its own learning rate, which are adjusted independently [Kingma and Ba, 2014].

3.2.2 Value Function Approximation

Generally, in order to compute the state-value function v^{π}, some form of iterative method is used, where the initial estimation of v^{π} is chosen arbitrarily, after which it is iteratively updated and improved upon. Primarily, the value function can be computed in two ways: Model-based or model-free. Model-based methods primarily rely on planning, which requires complete knowledge of the environment. Model-free methods, on the other hand, use only experience gathered from interaction with the environment. As such, these methods rely solely on learning and do not require a complete model of the environment's dynamics [Sutton and Barto, 2018, p. 159]. Model-free methods are in the following considered.

Model-Free Prediction

Let V denote an estimate of v^{π}. In general, an estimate is updated by finding the error between the current estimate and a target, which is some form of actual return G. At one end of the spectrum, a one-step return can be used. Here, one waits until the next time step $t+1$, and use the observed reward r_{t+1} as the target plus the discounted estimated value of the next state [Sutton and Barto, 2018, p. 143]:

$$G_{t:t+1} = r_{t+1} + \gamma \cdot V_t(s_{t+1}) \tag{3.2.12}$$

Let α be the step size of an update. Then using Equation (3.2.12), the update of an estimate can be expressed as [Sutton and Barto, 2018, p. 120]:

$$V(s_t) \leftarrow V(s_t) + \alpha \cdot \delta_t \tag{3.2.13}$$

$$\delta_t = r_{t+1} - \gamma \cdot V(s_{t+1}) - V(s_t) \tag{3.2.14}$$

This is known as a one-step Temporal Difference (TD) update. The update is said to *bootstrap*, which means its estimate is in part based on another estimate (in this case, $V_t(S_{t+1})$). δ_t is called the TD error, as it quantifies an error between the two estimates [Sutton and Barto, 2018, p. 121]. As the TD error is dependent on the next state and next reward, the TD error at time step t is first available one time step later, $t+1$.

At the other end of the spectrum one could instead wait until the complete return for a whole episode of length T is known [Sutton and Barto, 2018, p. 143]:

$$G_t = r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + ... + \gamma^{T-t-1} \cdot r_T \tag{3.2.15}$$

Using Equation (3.2.15) leads to the following update rule [Sutton and Barto, 2018, p. 119]:

$$V(s_t) \leftarrow V(s_t) + \alpha[G_t - V(s_t)] \tag{3.2.16}$$

This is known as a *Monte Carlo update*. However, instead of having to pick between two extremes – a TD-update or a Monte Carlo update – there is also an intermediate approach, where an update is based on an intermediate *n-step* return [Sutton and Barto, 2018, p. 143]:

$$G_{t:t+n} = r_{t+1} + \gamma \cdot r_{t+2} + ... + \gamma^{n-1} \cdot r_{t+n} + \gamma^n \cdot V_{t+n-1}(s_{t+n}) \tag{3.2.17}$$

Equation (3.2.17) can further be generalized to the case of any average of n-step returns, called the λ-return [Sutton and Barto, 2018, p. 290]:

$$G_t^{\lambda} = (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} \cdot G_{t:t+n} + \lambda^{T-t-1} \cdot G_t \tag{3.2.18}$$

Equation (3.2.18) is the most generalized form of the return, as it includes all the previously described ways in which the return can be calculated. For $\lambda = 1$, the sum reduces to zero, and the expression becomes a Monte Carlo update. For $\lambda = 0$, the expression reduces to $G_{t:t+1}$ and it becomes a one-step TD-update [Sutton and Barto, 2018, p. 290]. In summary, the state-value function estimate can in general be updated with Equation (3.2.18) as its target, where λ is set according to one's needs.

Objective Function

By using an MLP for function approximation, the approximate state-value function is parameterized by the parameter $\boldsymbol{\theta}$ [Sutton and Barto, 2018, p. 197]:

$$\hat{v}(s;\boldsymbol{\theta}) \approx v^{\pi}(s) \tag{3.2.19}$$

Approximating a state-value function is a regression problem, as the state-value function produces a numerical value. As a result, the objective function of the MLP is the MSE, Equation (3.2.10), with the λ-return G_t^{λ}, Equation (3.2.18), as target. Let J denote an objective function. Then the parameter update for $\boldsymbol{\theta}$ using gradient ascent with a learning rate α can be expressed as [Sutton and Barto, 2018, p. 321]:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \cdot \nabla J(\boldsymbol{\theta}_t) \tag{3.2.20}$$

3.2.3 Policy Gradient Methods

Policy gradient methods aims to iteratively optimize a *parameterized policy* π by the use of gradient ascent. If an MLP is used for function approximation, then the policy is parameterized by the parameter θ. With a parameterized policy, the probability that action a is taken at time step t, given state s, can be expressed as [Sutton and Barto, 2018, p. 321]:

$$\pi(a|s;\boldsymbol{\theta}) = P(A_t = a|S_t = s;\boldsymbol{\theta_t} = \boldsymbol{\theta}) \tag{3.2.21}$$

For a parameterized policy, its output is a Multinoulli probability distribution of the action preferences. In this regard, the softmax function is used: [Sutton and Barto, 2018, p. 322]:

$$\pi(a|s;\boldsymbol{\theta}) = \frac{e^{h(s,a;\boldsymbol{\theta})}}{\sum_b e^{h(s,b;\boldsymbol{\theta})}} \tag{3.2.22}$$

where $h(s,a,\boldsymbol{\theta})$ is the numerical preferences for each state-action pair. Actions with a higher preference in each state naturally have a higher probability of being selected. The denominator of Equation (3.2.22) ensures that all action probabilities in each state sum to one. By using a parameterized policy with soft-max action preferences, the approximate policy can both approach a deterministic policy or a stochastic policy [Sutton and Barto, 2018, p. 322-323].

There exist different ways in which the policy gradient can be computed. In its most general form, it can be expressed as [Schulman et al., 2015, p. 2]:

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \Psi_t \nabla \log \pi(a_t|s_t;\boldsymbol{\theta})\right] \tag{3.2.23}$$

$$\nabla \log \pi(a_t|s_t;\boldsymbol{\theta}) = \frac{\nabla \pi(a_t|s_t;\boldsymbol{\theta})}{\pi(a_t|s_t;\boldsymbol{\theta})} \tag{3.2.24}$$

where Ψ_t can have several expressions. Intuitively, a step in the policy gradient's direction increases the probability of taking an action better than the average, and decreases the probability of taking an action worse than the average. The gradient in Equation (3.2.24) is called the *eligibility vector*, and is the direction in the parameter space which yields the highest increase in probability of repeating action a_t for future visits to state s_t [Sutton and Barto, 2018, p. 327].

REINFORCE Update

In its simplest form, Ψ_t of Equation (3.2.23) can be the return G_t. This yields the following update of the parameter $\boldsymbol{\theta}$ [Sutton and Barto, 2018, p. 327]:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \cdot G_t \cdot \nabla \log \pi(a_t|s_t;\boldsymbol{\theta}_t) \tag{3.2.25}$$

Equation (3.2.25) is known as a REINFORCE update, and updates the parameter vector in the direction of actions that provide the highest return. As the REINFORCE update uses the complete return from time step t, it is a Monte Carlo update, as updates are made retrospectively after an episode has ended [Sutton and Barto, 2018, p. 327].

In some cases, all the actions, for a given state, may all have high values. For such instances, it can be difficult to distinguish which ones are the better among them. As such, a baseline can be introduced, as a means to identify the actions with the highest values. Likewise, a baseline can also be used for cases, where all actions have low values for a given state [Sutton and Barto, 2018, p. 329]. With a baseline, Ψ_t of Equation (3.2.23) can be expressed as the difference between the return G_t and a general baseline $b(s_t)$. The REINFORCE update with a general baseline is, as a result:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \cdot (G_t - b(s_t)) \cdot \nabla \log \pi(a_t|s_t;\boldsymbol{\theta}_t) \tag{3.2.26}$$

Reinforce updates have shown to produce estimates of high variance. As a consequence, learning with REINFORCE updates tends to be slow. Fortunately, an appropriate baseline, can mitigate this issue by reducing variance, and thus, accelerate learning.

Actor Critic Methods

An estimate of the state-value function can be used as a baseline. In particular, a learned value function, as described in Section 3.2.2, can be used. Methods that learn function approximations to both a parameterized policy $\pi(a|s;\boldsymbol{\theta})$ with parameter $\boldsymbol{\theta}$ and a parameterized state-value function $\hat{v}(s;\boldsymbol{\phi})$ with parameter $\boldsymbol{\phi}$, are called actor-critic methods. The learned policy is referred to as the *actor* and the learned state-value function is referred to as the *critic*. As the learned state-value function produces its value estimate through bootstrapping, where a value estimate is based on another estimate, it introduces a bias and a dependence on the function approximation's quality. In this case, the bias is actually beneficial to the policy update, as it reduces variance. As a result, actor-critic methods have faster learning, which also makes it more suitable for online learning or for continuing problems. An actor-critic policy update based on the λ-return, Equation (3.2.18), is as follows [Sutton and Barto, 2018, p. 321, 332-333]:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \cdot (G_t^{\lambda} - \hat{v}(s_t)) \cdot \nabla \log \pi(a_t|s_t;\boldsymbol{\theta}_t) \tag{3.2.27}$$

Generalized Advantage Estimation

So far, it has been established that REINFORCE updates have a high variance, resulting in slow learning. A way to mitigate this issue, is by the use of actor-critic methods that bootstrap. This introduces bias which yields lower variance, and thus facilitates faster learning. High variance results in slower learning and subsequently the need for more

samples. However, bias can be even more counterproductive, as it can cause learning to fail to converge or converge to a sub-optimal local optimum [Schulman et al., 2015, p. 1]. Bearing this in mind, what is sought, is a policy gradient method which makes a compromise between bias and variance. Generalized Advantage Estimation (GAE), proposed by Schulman et al. [2015], is a method designed to achieve exactly this. GAE makes use of an *advantage function* $A^{\pi}(s_t, a_t)$, which measures if the taken action a_t in state s_t is better or worse than average for that given state [Schulman et al., 2015, p. 2-3]:

$$A^{\pi,\gamma}(s_t, a_t) = Q^{\pi,\gamma}(s_t, a_t) - V^{\pi,\gamma}(s_t) \tag{3.2.28}$$

where:

- Q estimate of the discounted action-value function q^{π}, Equation (3.1.5)
- V V is an estimate of the discounted state-value function v^{π}, Equation (3.1.4)
- γ is the discount factor

Equation (3.2.28) reduces variance by utilizing the discount factor γ as a variance reduction parameter. The reduced variance, however, is at the cost of introducing bias. In close analogy to the λ-return Equation (3.2.18) the generalized advantage estimator GAE(γ, λ) of n-steps can be expressed as an exponentially-weighed average [Schulman et al., 2015, p. 5]:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{n=0}^{\infty} (\gamma\lambda)^n \delta_{t+n} \tag{3.2.29}$$

where $\gamma \in [0, 1]$ and $\lambda \in [0, 1]$. The two parameters, γ and λ both contribute to the bias-variance trade-off that occurs when using an approximate value function. The parameter γ determines the scale of the value function $V^{\pi,\gamma}$. which means if $\gamma < 1$, it introduces bias into the policy gradient estimate, independent of the value function's accuracy. However, if $\lambda < 1$, bias is only introduced when the value function is inaccurate. The optimal value of λ usually lies lower than the optimal value of γ. As such, λ presumably introduces less bias than γ [Schulman et al., 2015, p. 5].

Similar to the λ-return, there are two notable special cases for Equation (3.2.29), namely when $\lambda = 0$ or $\lambda = 1$ [Schulman et al., 2015, p. 5]:

$$\mathrm{GAE}(\gamma, 0) : \hat{A}_t = \delta_t = r_{t+1} - \gamma \cdot V(s_{t+1}) - V(s_t) \tag{3.2.30}$$

$$\mathrm{GAE}(\gamma, 1) : \hat{A}_t = \sum_{n=0}^{\infty} \delta_{t+n} = \sum_{n=0}^{\infty} \gamma^n \cdot r_{t+n} - V(s_t) \tag{3.2.31}$$

As seen in Equation (3.2.30), when $\lambda = 0$, the advantage reduces to a TD-error. As a result, it has a lower variance, but at the cost of a higher bias. On the other hand, as seen in Equation (3.2.31), when $\lambda = 1$, it becomes a sum of TD-error terms. As a consequence, it has a lower bias but in turn a higher variance. As a result, when $0 < \lambda < 1$, a compromise is made between bias and variance [Schulman et al., 2015, p. 5].

Using Equation (3.2.29) the policy gradient can be expressed as [Schulman et al., 2015, p. 5]:

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} \cdot \nabla \log \pi\left(a_t|s_t;\boldsymbol{\theta}\right)\right] \tag{3.2.32}$$

Design and Development 4

This chapter utilizes the previously described topics within Reinforcement Learning and the Flatland environment to formulate a design to base the development of the implementation upon. To achieve this, two overall parts need to be developed, namely a RL-model and an observation model. The RL-model's overall responsibility is to enable and optimize the agent's learning via a policy. The observation model's purpose is to provide beneficial input to the RL-model about the environment with which it interacts. The optimization of the observation enables the model to potentially make more informed and better decisions. These two coherent parts will then in combination construct a TMS for the Flatland environment.

4.1 Reinforcement Learning Model

Looking at the previously implemented RL-models in prior iteration's Flatland, they consist of both on-policy and off-policy based methods: *Proximal Policy Optimization* (PPO), *Ape-X* (Deep Q-Learning) and *Asynchronous Advantage Actor Critic* (A3C). These learning algorithms showed little performance difference between them [Laurent et al., 2021, p. 16]. Additionally, two contenders of the top three used actor-critic methods, respectively PPO and A3C, as their RL-model of choice. RL-models such as A2C & A3C, PPO and DDQN (Double Dueling Q-Network) were compared through experiments in ATARI cf. Hare [2019], which further cemented that the algorithms did not have a significant performance difference. However, PPO generally slightly outperformed the others in sample efficiency – the algorithm's ability to get the most out of a sample [Hare, 2019, p.36-45]. The ATARI experiments conducted by Schulman et al. [2017] shows that PPO also outperforms A2C, and performs similarly as, the recognized sample efficient RL-model, ACER, in terms of sample efficiency [Schulman et al., 2017, p. 1].

Based on these findings, the focus when selecting a RL-model is specifically on ease of implementation and hyperparameter tuning, and scalability. PPO offers a code-simplified alternative to ACER and Trust-Region Policy Optimization (TRPO), which are recognized to be highly sample efficient [Schulman et al., 2017, p. 1]. This was base for further improvement of the PPO model implemented in Flatland.

4.1.1 Proximal Policy Optimization

Vanilla policy gradient methods fall short in mainly three areas. First, they can be difficult to scale to larger environments. Second, they can be challenging to apply to a variety of different problems without requiring hyperparameter tuning. Third, they have a poor sample efficiency, and can require millions or billions of time steps to learn simple tasks

[OpenAI, 2021a; Schulman et al., 2017, p. 1]. Proximal policy optimization, proposed by Schulman et al. [2017], is a prevalent approach for addressing these shortcomings.

The primary fallibility of vanilla policy gradient methods is too large policy updates, as they accidentally lead to a suboptimal policy in the long run. As such, there needs to be a constraint on the size of a policy update, in order to prevent too large policy updates. PPO solves this by using the following objective function [Schulman et al., 2017, p. 2-3]:

$$J^{CLIP}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta) \cdot \hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \cdot \hat{A}_t \right) \right] \tag{4.1.1}$$

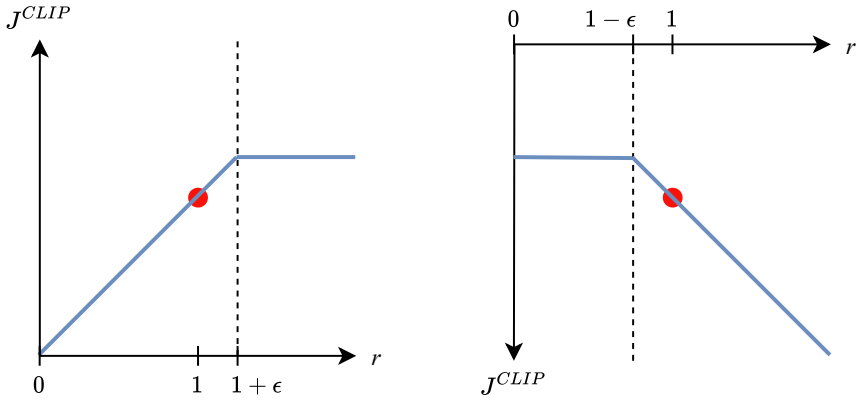
where:

- r_t The probability ratio: $r_t(\theta) = \frac{\pi(a_t|s_t;\theta)}{\pi_{\text{old}}\ (a_t|s_t;\theta)}$
- $\hat{A}_t$ The advantage estimate at time t
- ϵ A hyperparameter for limiting the size of a policy update

To provide an easier interpretation of Equation (4.1.1), it is rewritten as follows [OpenAI, 2021b]:

$$J^{CLIP}(\theta) = \begin{cases} \min(r_t, 1+\epsilon) \cdot \hat{A}_t, & \hat{A}_t \geq 0 \\ \max(r_t, 1-\epsilon) \cdot \hat{A}_t, & \hat{A}_t < 0 \end{cases} \tag{4.1.2}$$

As seen in Equation (4.1.2), when the advantage is positive, the objective will increase if the probability of taking the current action, $\pi(a_t|s_t;\theta)$, is increased. However, the minimum puts a ceiling on how much the objective can increase. The objective cannot grow beyond, $(1+\epsilon) \cdot \hat{A}_t$, as soon as, $\pi(a_t|s_t;\theta) > (1+\epsilon) \cdot \pi_{\text{old}}(a_t|s_t;\theta)$. Conversely, when the advantage is negative, the objective will increase if the probability of taking the current action, $\pi(a_t|s_t;\theta)$, is decreased. Similarly, the maximum puts a limit on how much the objective can increase. In particular, it cannot increase beyond, $(1-\epsilon) \cdot \hat{A}_t$, as soon as, $\pi(a_t|s_t;\theta) < (1-\epsilon) \cdot \pi_{\text{old}}(a_t|s_t;\theta)$ [OpenAI, 2021b]. Figure 4.1 illustrates how J^{CLIP} constrains objective function updates.



(a) $A \geq 0$ **(b)** $A < 0$

Figure 4.1: The objective function J^{CLIP}, of a single time step, as a function of the probability ratio r. (a) represents J^{CLIP} when the advantage is positive. (b) represents J^{CLIP} when the advantage is negative. The red circle represents the starting point, $r = 1$, for the optimization. The figure has been inspired by [Schulman et al., 2017, p. 3].

Besides Equation (4.1.1), Schulman et al. [2017] also proposes another way to constrain policy updates by the use of a KL-penalized objective function. However, as the J^{CLIP} outperforms this approach [Schulman et al., 2017, p.4-5], it is not taken into further consideration.

With an actor-critic architecture, where the actor and critic share parameters among them, Equation (4.1.1) has to be modified further, such that the objective function also includes a value function error term. Alongside this, Equation (4.1.1) can additionally be augmented by adding an entropy bonus term to incentivize adequate exploration. Adding the aforementioned two terms to Equation (4.1.1) constitutes to the following objective function [Schulman et al., 2017, p. 5]:

$$J_t^{CLIP+VF+S}(\theta) = \mathbb{E}_t \left[J_t^{CLIP}(\theta) - c_1 \cdot J_t^{VF}(\theta) + c_2 \cdot S\left[\pi_\theta\right](s_t) \right] \tag{4.1.3}$$

where:

S	Entropy bonus function
c_1	Value function coefficient
c_2	Entropy bonus function coefficient
J_t^{VF}	MSE between value function estimate and target

Generally, in RL, the agent strives to maximize its reward. Maximization is achieved by minimizing the negative objective function:

$$-J_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[-J_t^{CLIP}(\theta) + c_1 \cdot J_t^{VF}(\theta) - c_2 \cdot S\left[\pi_\theta\right](s_t) \right] \tag{4.1.4}$$

MAPPO

There are other variants of PPO that seek to solve the Multi-Agent RL (MARL) problem, namely, Multi-Agent PPO (MAPPO). Comparing to standard PPO (or Independent PPO in Yu et al. [2021] for the sake of comparison), MAPPO follows the algorithmic structure of standard PPO. However, the MAPPO variant differs in terms of observation scope in the state-value function estimate, also called *centralized* or *decentralized* policy gradient methods [Yu et al., 2021, p. 1-3]. Recall that, PPO has an actor-critic architecture; the actor is a learned policy function π_θ and the critic is a learned state-value function $\hat{v}_\theta(s)$. With MAPPO, $\hat{v}_\theta$ is used for variance reduction and is only used for training. Therefore, it can take extra global information, by utilizing one of three centralized value function inputs: *Concatenations of Local Observations* (CL), *Environment-Provided Global State* (EP) and *Agent-Specific Global State* (AS). AS mixes CL and EP, concatenating both local Observations and global States. Yu et al. [2021] feature prunes the overlaps when mixing the two, using a *Feature-pruned Agent-Specific Global State* (FP) [Yu et al., 2021].

At first glance, it seems enticing to utilize this PPO variant, MAPPO, since it fits the MARL problem that is prominent in the Flatland challenge. Yu et al. [2021] finds that MAPPO performs better than the state-of-the-art off-policy methods in terms of sample efficiency in their experiments. However, the findings also show that the MAPPO's performance is very similar to standard PPO [Yu et al., 2021]. Another point, that diminishes the appeal of implementing MAPPO, is the time to implement vs return trade-off. All else unchanged, MAPPO will take longer to implement than standard PPO,

since the model has a complex interplay between the critic and observation model of the environment. Due to the performance not showing clear signs of improvements comparing to standard PPO cf. Yu et al. [2021] and the model's complexity of implementation, it is deemed insufficient for the Flatland case, which is why it will not be taken into further consideration.

4.2 Design Overview

On the whole, a TMS for the Flatland environment can be viewed as a MARL problem. With this view, an RL-based TMS is a decentralized system, consisting of a network of several independently acting collaborative agents. There exists many ways in which an RL-based TMS can be realized. For instance, the RL-based TMS can consist of decentralized learned policies, where each agent has its own policy. On the other hand, the RL-based TMS could also only utilize a single centralized learned policy, that all agents make use of. This project considers the latter. As such, each TMS agent acts individually in the Flatland environment, but makes use of the same policy, which has been learned in a centralized manner using PPO. An overview of the components of each TMS agent is shown in Figure 4.2.

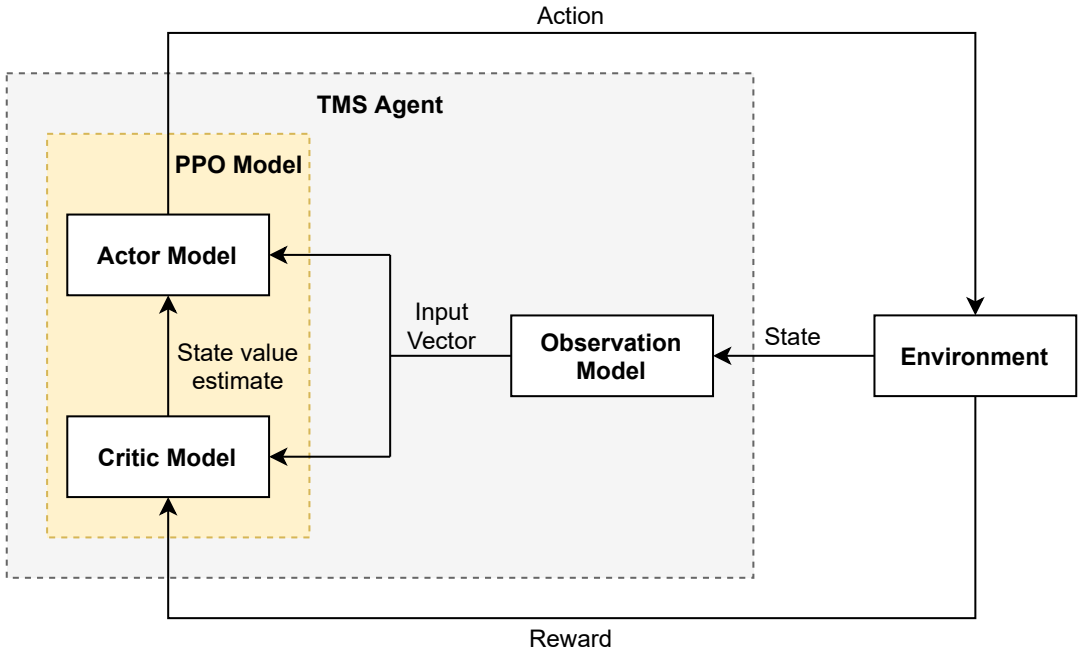


Figure 4.2: The components of a TMS agent. Each TMS agent consists of two components: An observation model and a PPO model. The observation model's purpose is to process the environment and provide the PPO model with the necessary information needed, in order to learn and make sensible actions. The PPO model has the purpose of providing the TMS agent with a learned policy, in which it can base its actions on.

4.3 PPO Model

The PPO model employs an actor-critic architecture. As such, it consists of two MLPs: An actor MLP with parameter $\boldsymbol{\theta}$ and a critic MLP with parameter $\boldsymbol{\phi}$. The actor MLP outputs a learned policy. There are five different actions that an agent can do. Correspondingly, the actor MLP outputs soft-max action preferences for the five existing actions. The critic MLP, on the other hand, outputs a learned value function. For this reason, it has a linear output activation function, as it deals with a regression problem. The network architectures of the actor MLP and critic MLP are shown in Figure 4.3 and 4.4, respectively.

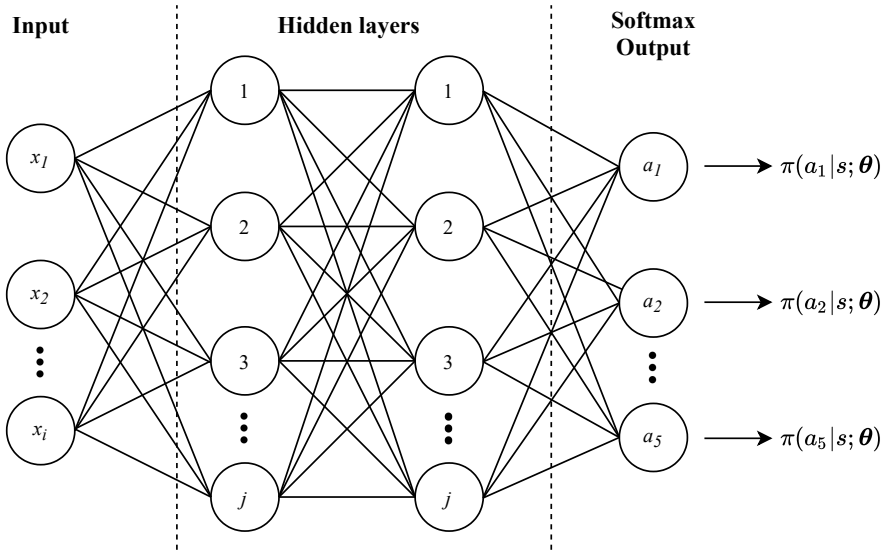


Figure 4.3: The actor model's network architecture. The actor takes an input vector, processes it with its hidden layers and uses a softmax function to produce a Multinoulli distribution for the five existing actions.

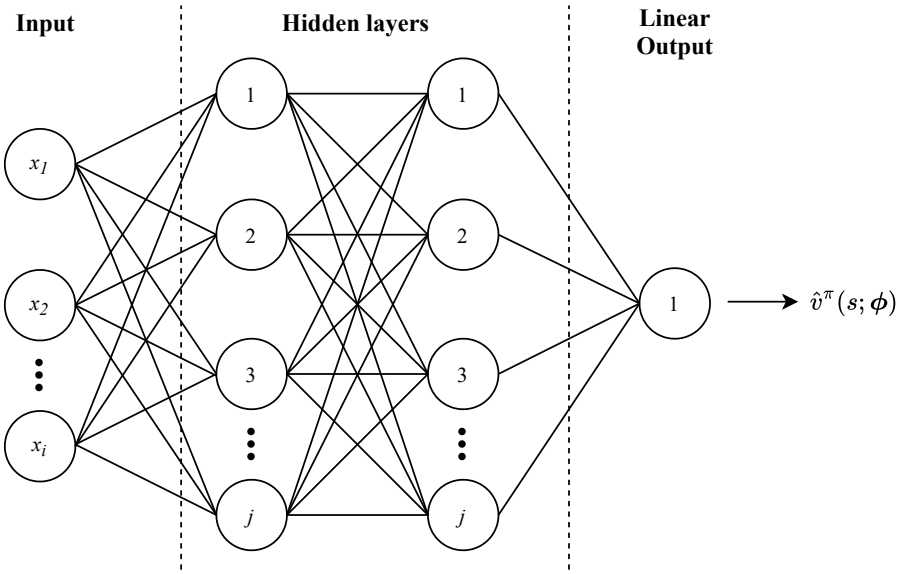


Figure 4.4: The critic model's network architecture. The critic takes an input vector, processes it with its hidden layers and outputs a state value estimate.

Learning

All agents make use of a centralized learned policy. The PPO model has been designed such that n agents run the learned policy for T time steps, where all experience is gathered in a central experience buffer. The collected experiences are used to update the policy, after which the experiences are discarded. In other words, policy updates are made in a retrospective manner. An experience consist of the following:

- State s_t
- Action a_t
- Reward r_t
- Next state s_{t+1}
- Done: when a terminal state is reached for the agent.
- Critic MLP's state value estimate $\hat{v}(s_t; \boldsymbol{\phi})$

The whole learning process is illustrated in Figure 4.5.

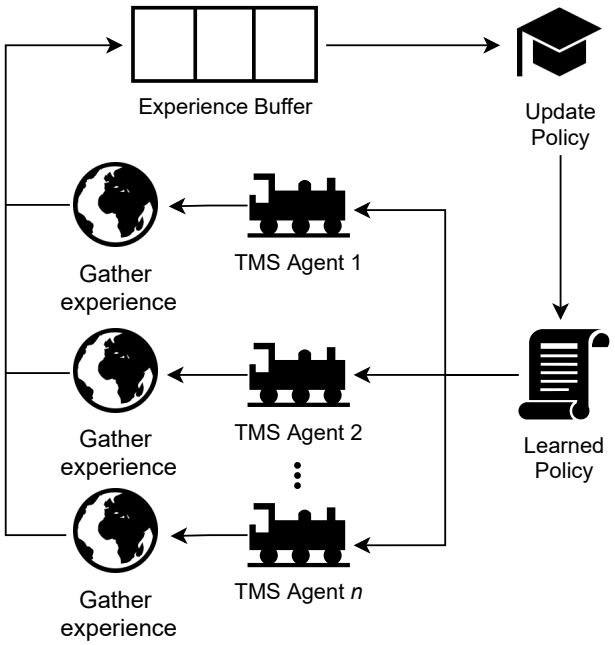


Figure 4.5: Conceptualization of learning for a centralized policy. During learning all the TMS agents use the same policy to gather experience for T time steps, whereafter these experiences are used retrospectively to improve upon the policy.

The actor MLP and critic MLP have been designed to share parameters between them. For this reason, both of them use Equation (4.1.4) as their objective function. When learning, the actor MLP and critic MLP have different goals in mind. For the actor MLP, a step is taken in the direction which increases the probability of taking better actions, and decreases the probability of taking inferior actions. For the critic MLP, a step is taken in the direction which minimizes the MSE between its state value estimate and the actual return.

As mentioned in Section 3.2.3, the central idea of actor-critic methods, is to make use of some form of bootstrapping estimate to assist the learning of a policy. In this regard, PPO

utilizes a GAE, as shown in Equation (4.1.1), which, roughly speaking, measures if a taken action is better or worse than average for that given state. GAE makes use of the critic MLP's state value estimates, namely $\hat{v}(s_t; \boldsymbol{\phi})$ and $\hat{v}(s_{t+1}; \boldsymbol{\phi})$, to compute its estimates:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \delta_t + (\gamma \cdot \lambda) \cdot \delta_{t+1} + ... + (\gamma \cdot \lambda)^{T-t+1} \cdot \delta_{T-1} \tag{4.3.1}$$

$$\delta_t = r_t + \gamma \cdot \hat{v}_t(s_{t+1}; \boldsymbol{\phi}) - \hat{v}_t(s_t; \boldsymbol{\phi}) \tag{4.3.2}$$

Optimization Algorithm

As mentioned in Section 3.2.1, an Adam optimizer is currently a favored optimization algorithm, and is suitable for non-stationary objective functions. For these reasons, an Adam optimizer is used as the optimization algorithm in this project. In order to obtain an unbiased estimate of the gradient, when using a stochastic gradient method, the experiences from the experience buffer needs to be drawn i.i.d. As such, the experience buffer is shuffled at the beginning of each epoch.

4.4 Observation Model

Building upon the initial definition of the Flatland environment and presented design overview of PPO this section is devoted to constructing an observation model. The goal is to provide the TMS agent with a fitting set of information, which enables the agents to reason about their environment.

In order to achieve this, it is required to select an observation method, that can observe the Flatland environment. Subsequently, the features of the observation method needs to be determined. Features are functions of the states in the environment that return values describing the states [Poole and Mackworth, 2010, p. 112]. The features of the observation model provides input to the agents, as it enables the agents to use the information encoded by the features to reason about the states in the environment. This will ultimately enable the agents to learn an improved policy π, which can be applied to future states. Having stated the goal and the overall tasks of constructing an observation model for the Flatland environment, the next section is devoted to choosing an appropriate observation method and a set of features, as well as describing them in depth.

4.4.1 Observation Method

As presented in Section 2.2, Flatland provides three default observation methods: Global, local, and tree observation (See Figure 2.3). By examining the tree observation in detail, it is a quadtree data structure as described in Section 2.2, and it is the method utilized by most RL solutions - including the winning solutions from the previous iteration [Laurent et al., 2021, p. 7]. It utilizes the underlying graph structure of the railway network, where each switch or dead end correspond to a node and a branch correspond to a path between two nodes. For instance, if an agent's current position is located at a switch, two transitions are possible. From each allowable transition, a branch will then expand to the two upcoming nodes, which contains information gathered along the branches to the nodes. This information is calculated by 12 default features which Flatland provides [AIcrowd, 2021h]. The tree data structure, combined with the 12 features, provides the agent with information about the states of the environment.

Examining the two other observation methods, global and local observation, they are essentially the same observation method with local being a subset of the global observation, as described in Section 2.2. The global observation is as the name implies a global view of the grid. It is encoded as $h \times w \times c$. h is the height, w is the width of the environment, and c is the number of channels or features describing the states of the environment, as accounted for in Section 2.2. It includes a transition map, which stores a unique value for each type of transition and its corresponding direction. The transition map contains information on which cells have a transition, and if that is the case what types of transitions are allowed. By closer inspection of the features in the method, it reveals they are more focused on a global aspect, rather than providing detailed information about the surrounding states from an agents' current state. It includes features such as all agent's position, direction, and speed.

This makes the global method less suitable for the given developed PPO; while the agents act from the same policy, they make choices individually without communicating with each other. This fact gives an inherent advantage to the tree observation in this context, as it can see several decision points ahead for each agent, in a much more efficient way than the global observation. Imagine having an environment with 50 agents with a global observation method and looking three decision points ahead. The number of possible variations in regard to the locations of the agents is not feasible to represent as global observations, because each observation takes up so much space.

However, if the implementation had fallen on a centralized critic following the MAPPO methodology as described previously, instead of having each agent act independently, the global observation would have made more sense 4.1.1. In this case, where a centralized "watch tower" acts on behalf of the agents, there is no need to see as many decision points ahead for the agents individually, as the centralized agent can see all agents at once, making coordination between agents much better.

This means that the global observation is not worse at solving multi-agent navigation problems than the tree observation, but more suited to another type of PPO implementation, which is the reasoning for going with the tree observation in this project.

4.4.2 Tree Observation

Now that the choice has settled on the tree observation, this observation model and its features will be explored more in depth. Every node of the tree has some defined features, which provides information to the agents about the environment as well as the other agents. The baseline features are as follows:

- *Distance to agent's target encountered*
- *Distance to other agent's target encountered*
- *Distance to other agent encountered*
- *Distance to potential conflict*
- *Distance to next unusable switch*
- *Distance to next usable switch*
- *Minimum distance to target*
- *Number of agents with same direction*
- *Number of agents with opposite direction*
- *Malfunctioned agent encountered*
- *Minimum fractional speed*
- *Number of agents ready to depart*
- *child nodes*

All the nodes contains the above listed features, which are filled with information gathered from the path to the next node. *Encountered* means that a given attribute is within an agent's observation. Most of the features are intuitive from their description, except two:*Malfunctioned agent encountered* and *Minimum fractional speed*. The feature *Malfunctioned agent encountered* stores the number of time steps an agent is malfunctioned, if a malfunctioned agent is detected on the searched path. The feature *Minimum fractional speed* stores the slowest observed speed of an agent with the same direction on the searched path [AIcrowd, 2021h].

The most essential feature in regard to the structure of the tree observation is the *child nodes* feature, which defines the child nodes that are built from a node's possible transitions. The tree builds a child node for each of the actions an agent can take i.e ['Left', 'Forward', 'Right', 'Backwards'], but this is only done if there is more than one possible transition that the agent can take, given its current position and orientation. If only one transition e.g. forward is valid, it will return [0, 1, 0, 0], and it will not check the other actions, as this would be unnecessary. In the case where only one transition is valid no node is built, and the action here would be either forward or no-operation. However, if there is more than one valid transition the agent checks all directions for valid transitions. For each valid transition, it builds child nodes recursively until it reaches the defined max depth of the tree, which in our case is two.

The observation space using a tree structure with a depth of two is illustrated in Figure 4.6. which is a snapshot of the rendered Flatland environment. With a tree depth of two, it has two decision points ahead. The first decision point in this scenario is at `blue circle number one`, where the agent can choose to turn right or continue straight. As it has two different actions it can take at step one, it has to make two predictions at step two. If it chooses to go right, it will go to `decision point 2.1`. On the other hand, if

it goes straight, it will go to `decision point 2.2`. Then depending on the action that is executed, the agent either go to `blue circle 2.1` or `blue circle 2.2`, and this node will become the new root node, where a new decision needs to be taken from.

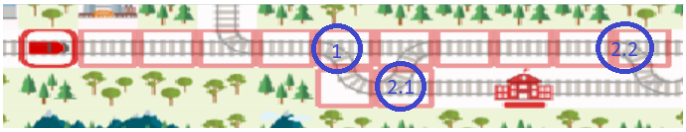


Figure 4.6: Tree observation Flatland. The red squares are the parts of the railway that the agent can observe and the blue circles are nodes.

This also exemplifies why it is costly to increase the tree depth. If a tree depth of three instead was applied, then the method would have to calculate predictions for all the decision points at level two, which for each node would result in an additional two nodes. Then, if the depth was increased to four, the method would have to make predictions for all decision points at level three, which would result in eight nodes. This means that the time complexity of the tree structure is $\Theta(2^n)$, where n is the depth of the tree. Therefore, thorough considerations have to be made between computational complexity and information gain when expanding the depth of the tree.

After the observation method has explored the different actions an agent can take, the tree has to be transformed to an appropriate format in order to be processed by the PPO-model. The transformation process is illustrated in Figure 4.7.

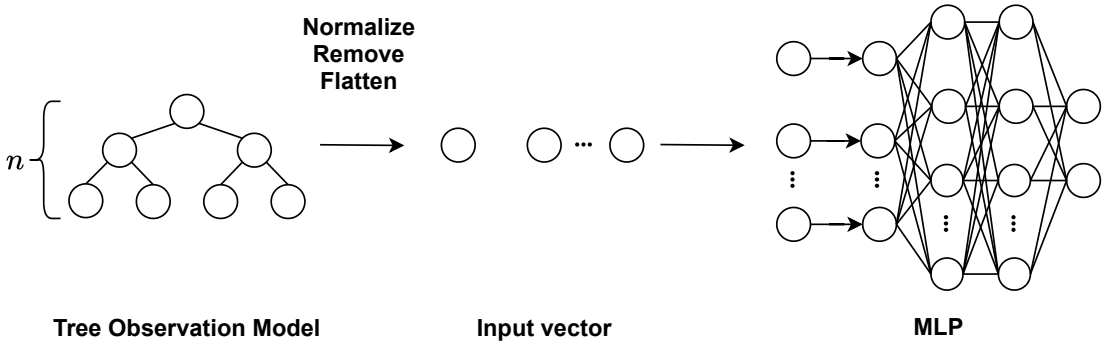


Figure 4.7: Conceptualization of the data preparation process. The tree observation model is normalized to the range [0,1], non-instantiated features are removed, and the tree observation model is flattened to a vector.

Initially, when the observation is returned, it is a multidimensional tuple consisting of nodes corresponding to the depth of the tree. Each node has numerical values for all features if they are instantiated, otherwise they contain infinity values. Each node also contains sub nodes for each action which is valid as well.

The input format of the PPO model is a vector of normalized values. First, the values of the tuple are normalized to values between zero and one, this serves two purposes: to make values comparable, as well as removing features and sub nodes which contain infinity, as they are not instantiated. This is also the reasoning behind infinity values, as when they

are normalized they do not get a value between zero and one, and are thus easily detected and removed.

4.4.3 Features of the Tree Observation Method

Having provided a deeper understanding of the implementation of the tree observation, the focus will now be on selecting the features of the observation method. The default method contains 12 in-built features, which are listed in Section 4.4.2. Besides utilizing the default 12 features, an opportunity lies to implement additional features, to optimize the information input to the PPO model. Adding additional features is a trade-off between the computational cost and the performance improvement that the new features provides. The main focus is to implement features which enhance the coordination between agents, resulting in a higher reward gain. This is to mitigate the challenge of having multiple agents which needs to navigate the same environment simultaneously. Some consequences of insufficient coordination between agents could be deadlocks, late arrivals or not arriving at all. Thus, improving the coordination would theoretically result in maximization of rewards.

Given this premise, the following six features has been devised:

- *Excess time to target*
- *Total cells*
- *Number of crossings*
- *Number of merging rails*
- *Dispute next cell*
- *Has deadlocked agent*

In the next sections, the implementation and purpose of the additional features are expanded upon, as well as the purpose of the features. Subsequently, these features are going to be evaluated on in Section 5.3.3.

Excess Time to Target

The purpose of the feature *Excess time to target* is to provide the agents with information on the number of time steps the agents have in excess. This is the difference between the latest arrival time to a target location and the number of time steps, it takes to get from a position in the environment to the target location. The intuition with this feature is that rewards are based on compliance with the time tables. For this reason, it is useful for the agents to know how much time they have in excess to still be able reach the target on time.

Total Cells

The feature *total cells* calculate the number of cells between two nodes. The intuition for this feature is that the total cells until the next node is an indicator of the uncertainty associated with that path. If the *total cells* equals 100 then the agent is dependent on this decision for a longer amount of time than if the *total cells* equals 10. Note that, the uncertainty of a path is also affected by the number of crossings and the number of merging rails.

Furthermore, knowing the total cells on a path could possibly allow for easier overtakes - especially considering that the fractional speeds of other agents on the path is already known after an environment is instantiated. This is illustrated in Figure 4.8.

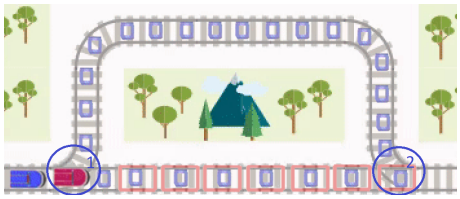


Figure 4.8: The first node is marked as the blue circle number one, where the blue train can choose to go left or continue forward behind the red train, to end up at the merging node marked as blue circle number two.

As an example, the blue agent's fractional speed, could be considerably higher then the red agent's. The path to the left is the longer path, as indicated by the 18 blue squares on the left path. Because the blue agent knows that the fractional speed of the red agent is significantly lower, and it knows the length of the left path, it should allow the agent to reason about the choice of taking the left path. This will allow the blue agent to overtake and not get stuck behind the red agent.

Number of Crossings

The feature *Number of crossings* calculates the number of rails which crosses the agent's path to the adjacent node. This is captured in Figure 4.9. As illustrated by the two possible paths, the red path has one crossing, where the blue path has three crossings. The purpose of this feature is to indicate the uncertainty associated with a given path. If an agent has two possible paths, the one path with the least number of crossings will generally be the one with the least probability of a deadlock.

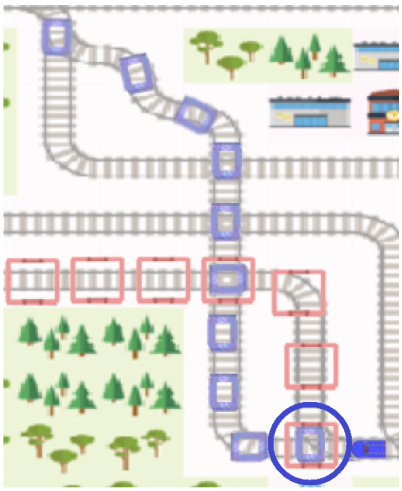


Figure 4.9: An agent is located to the right of the node with the blue circle. From here, the agent has two possible paths to the adjacent nodes. One path is denoted by the red squares and the other with blue squares.

Number of Merging Rails

The idea behind the feature *Number of merging rails* is very similar to the feature *Number of crossings*, which is to capture points where agents paths are overlapping with each other. The specific point this feature captures is where paths merge in the environment. The scenario is illustrated in Figure 4.10. The purpose of the feature is to capture the merging points, where the agent itself can not observe the other merging path, but receive information about the number of merging paths, to give an indication of the probability of a deadlock occurrence for a specific path. This is also illustrated in Figure 4.10, where the two agents potentially can end up in a deadlock.

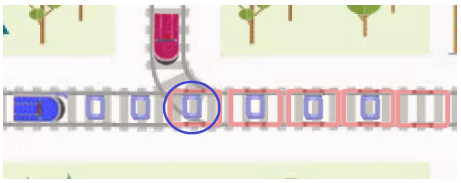


Figure 4.10: The illustrated scenario captures the points where two paths are merging. This is indicated by the blue circle. The blue agent can only observe the straight forward path, as the path which merges with the blue agents path, is not a valid transition for the agent itself. This is indicated by the blue squares, which are what the agent can observe.

Dispute Next Cell

The feature *Dispute next cell* looks at the scenarios where two agents' next positions are the same cell, meaning they will encounter a potential deadlock, if neither of the agents do not change their future action. This is illustrated in Equation 4.11. To mitigate this scenario, this feature aims to solve the conflict, by randomly assigning only one of the agents with the feature value, *Dispute next cell*. The intuition behind this choice is that the agents should learn if this feature is set to 1, then the agent needs to stop in order to avoid a deadlock. As such, the feature should remove some potential deadlocks, where agents' positions stay the same, since both agents are waiting for the other agents to move forward. However, there are scenarios that this feature cannot handle at all, such as when two agents are moving towards each other on a straight path, where there is no allowable switch for both agents.

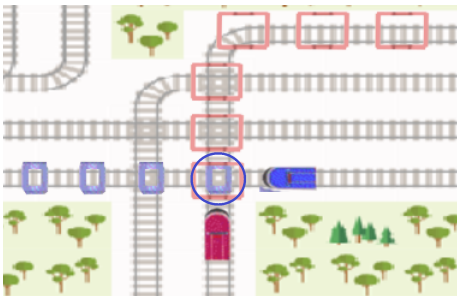


Figure 4.11: In this scenario, a red and blue agent are located in a cell right before a crossing, where both agents next position is the same, outlined by the blue circle. If not one of the agents changes their action from stop moving to move forward, they will be in a deadlock.

Has Deadlocked Agent

The new features were based on what was expected to improve the agents' decision-making. An important feature, in this aspect, is recognizing when a node contains a deadlocked agent. If an agent is at a switch where one path contains deadlocked agents, it is important that the agent chooses the other path, so it does not get deadlocked itself. If the agent chooses the direction that contains a deadlock, then the agent will almost certainly be deadlocked. However, for implementing such a feature, it is required to know which agents are deadlocked.

The intuition behind the following method, which returns the set of deadlocked agents, is based on three cases. If one of these cases is true, then the agent is deadlocked.

Case 1: Trying to exchange positions
If agent A is trying to reach agent B's current position and agent B is trying to reach agent A's current position, then they will be deadlocked, as they will drive directly into each other. This case is illustrated in Figure 4.12. The arrows in the figure represents the agent's desired next position.

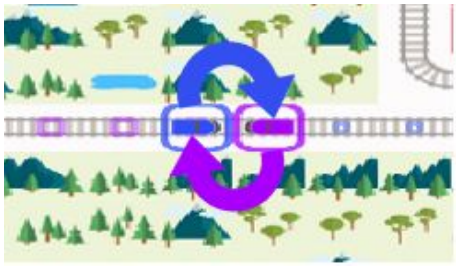


Figure 4.12: Illustration of two agents trying to exchange positions.

Case 2: Driving simultaneously into the same cell
If two agents are trying to drive simultaneously into the same cell, then one of the agents will drive into the other agent. This can both happen if the agents are driving directly towards each other, but also if they are driving into the same cell from different angles. This is illustrated in Figure 4.13.

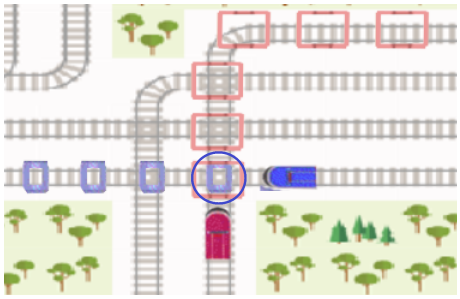


Figure 4.13: Illustration of two agents trying to reach the same cell simultaneously.

Case 3: Driving into a deadlocked agent
If an agent is trying to drive into a cell that is already occupied by a deadlocked agent, then the agent itself will be deadlocked. This is illustrated in Figure 4.14. The two left-most

agents in the figure are already deadlocked as they drove directly into each other. The two right-most agents in the figure will both be deadlocked too, as they are trying to drive into a cell that is already occupied by a deadlocked agent.

Figure 4.14: Illustration of agents trying to reach a cell that is occupied by a deadlocked agent.

4.5 Performance Optimization

Besides the implementation of the PPO model and the new features in the observation method, additional methods can potentially enhance the overall performance of the final configuration of the agent. This involves hyperparameter tuning, action-masking and frame skipping, which will be elaborated upon in the following sections.

4.5.1 Hyperparameter Tuning

A hyperparameter is parameter which influences the learning process. Therefore, tuning and these parameters can provide valuable benefits. Hyperparameter tuning is an optimization process, where appropriate hyperparameter values are determined with the aim of improving performance of a machine learning model. This section will describe the methodological approach to selection of hyperparameter ranges, which will be used for evaluation in Chapter 5.

As stated by OpenAI [2021a] following the release of the PPO-paper, PPO sought to strike *a "balance between ease of implementation, sample complexity, and ease of tuning"*. This is partially accomplished by reducing the amount of hyperparameters. In this project, the hyperparameters in table 4.1 will be utilized, in order to optimize the performance of the model.

The default values, which are used prior to the hyperparameter tuning, are gathered from previous Flatland submissions that uses PPO [mitchellgoffpc, 2020][adrianegli, 2020]. This ensures a better starting point for further improvement. The default values still coincide with parameter ranges used in recent PPO research [Henderson et al., 2018; Engstrom et al., 2020]. The search space is defined by adding additional values, so the specified values equally represent the outer bounds and middle of the range. Initially, the focus will be on cutting down the search space, to match the available computational resources. The focus on limitation of computational spending, will additionally translate to the size of the search space, which in possible future iterations would attract more attention. When defining the search space, no prior relation between parameters will be assumed, except for batch and buffer size, which are divisible with each other according to observations in recent papers.

Selecting Hyperparameter Search Spaces

- *Hidden layers*: Describes the amount of hidden layers, as the name suggest. The hidden layers are omitted from the chosen hyperparameters, due to sufficient results showcased throughout other research with two layers in smaller network architectures [Schulman et al., 2017; Henderson et al., 2018; Engstrom et al., 2020]. Thus, time can be saved by focusing on the remaining hyperparameters. Furthermore, it is chosen to stray away from different hidden layer sizes for the actor and critic, to reduce the total amount of sweeps required.

- *GAE parameter (λ):* Bias-variance trade-off parameter. It is fixed to 0.95 in effort to reduce computational resources and time. As [Schulman et al., 2017] have chosen to use this value, as well as previous Flatland submissions [adrianegli, 2020], it is presumed that the selected value generalizes well across environments.

- *Activation function:* Controls the activation of neurons throughout the model, initially it is set to Tanh as default, as this is used in the original [Schulman et al., 2017] and later papers. Some papers suggest that Tanh outperforms ReLU [Henderson et al., 2018], in that spirit, both activation functions are added to explore if this hypothesis holds true.

- *PPO clip factor (ϵ):* Constrains the size of a policy update for a PPO objective function. The clip factor adheres to the value range listed in the original paper: 0.1, 0.2, where it is assumed that the empirical testing is generalizable across configurations of PPO. Although the 0.3 clip is tested in the PPO paper, it was predominantly 0.2 clipping which performed better. The primary factor for this search space's existence is to see if the 0.2 results, from the original paper, is applicable to the Flatland environment [Schulman et al., 2017].

- *Buffer size:* The amount of experience gathered by n agents before a policy update is performed. Its search space defined by its relationship to the mini-batch size, such that they are divisible by each other. With its default value being 32,000, defined by $128 \cdot 250$. With 128 being the batch-size and 250 being the factor representing the relative size difference. The search space is defined by defining a series of arbitrary values timed with the mini-batch size, in this case 200, 250 and 300, varying with a factor of 50, to explore the effect of a higher and lower buffer sizes, comparatively to the default value(.

- *Mini-batch size:* The amount of samples which are forward passed in an MLP, before a backward pass is done. It is defined in powers of two, 2^n, in a range spanning from 64, which is the mini-batch size listed in the PPO paper, to the batch-size of 128 currently used, to a batch size of 256 to explore higher values effect on performance.

- *Discount factor (γ):* Determines the model's prioritization of immediate and future rewards. The search space defined by its default value. set from a previous Flatland submission, the original PPO papers default at 0.99 and an explorative value of 0.95, to see a lower discount factor's effect on performance.

- *Learning rate (α):* The step size of the gradient optimization method. It is set to $5 \cdot 10^{-5}$ as default based on previous submissions. The search space is scaled logarithmic, such that $5 \cdot 10^{-4}$ and $5 \cdot 10^{-3}$ are also included, with the latter bearing close resemblance to the $3 \cdot 10^{-4}$ which is used by Schulman et al. [2017]. Different learning rates for the actor and critic are omitted from the hyperparameter tuning in the model, to reduce the time spent hyperparameter tuning.

- *Hidden layer size:* The amount of neurons in each hidden layer. The search space is set to 64, 128 and 256. This is in an effort to explore the smaller hidden layer size, represented in the PPO and additional papers, as well as exploring larger network architectures and their correlation to overall performance.

- *Epochs:* Accounts for the amount of iterations done through the training-data, is set to the search space 3, 5 and 10. The amount of epochs in the search space does not exceed the current default value, as additional epochs would be presumed to have a negative impact on the time spent training, where training in some current instances takes approximately 18 hours.

- *PPO Value function coefficient (c_1):* Determines the influence of the value function error term in the PPO objective function, shown in Equation (4.1.3). It is set to 1 or 0.5 as all papers explored in relation to this project has had its values in either of the two values, which could indicate generalizability between different domains.

- *PPO entropy coefficient (c_2):* Determines the influence of the entropy bonus function, shown in Equation (4.1.3). The coefficient is set to 0 or 0.01 as all explored papers has had its values being either of the aforementioned, which should imply the generalizability across different domains.

In table 4.1, each of the aforementioned search spaces as well as default values can be seen pictured, which will be utilized in a hyperparameter sweep in Chapter 5.

Hyperparameter	Default value	Search space
Activation function	Tanh	[Tanh, RelU]
Buffer Size	32,000	[$b \cdot 200$, $b \cdot 250$, $b \cdot 300$]
Mini-batch Size	128	[64, 128, 256]
Discount (γ)	0.97	[0.95, 0.97, 0.99]
GAE parameter (λ)	0.95	[0.95]
Learning rate (α)	$5 \cdot 10^{-5}$	[$5 \cdot 10^{-5}$, $5 \cdot 10^{-4}$, $5 \cdot 10^{-3}$]
Hidden layer size	128	[64, 128, 256]
Number of hidden layers	2	[2]
PPO clip factor(ϵ)	0.1	[0.1, 0.2]
Epochs	10	[3, 5, 10]
Value function coefficient(c_1)	0.5	[0.5, 1]
Entropy Coefficient(c_2)	0.01	[0, 0.01]

Table 4.1: Table of hyperparameters with their default values and search spaces. b in the figure denotes mini-batch size

Selecting Hyperparameters

In order to optimize the selection process of hyperparameters, external libraries are used to manage and expedite the process. Weights and bias's (WandB) sweep tool is used [Wandb.ai, 2021], which gives fast access to the primary search strategies used to navigate and select through search spaces. In general, there exist three search space strategies:

- *Grid search*: Sets up a thorough walk-through of every combination of search space values. It is very effective, but can take an incredibly long time to train, which is not feasible in the context of this project[Yu and Zhu, 2020].

- *Bayesian Search:* Uses a Gaussian probabilistic model, to track the relationship between the selected metric and the hyperparameters and picks hyperparameters to increase the probability of increasing the models' performance[Yu and Zhu, 2020].

- *Random Search:* Selects combinations of search space values at random. This approach is not as efficient as grid search, but provides promising results with the use of far less computational resources, which is why it will be utilized for hyperparameter tuning[Yu and Zhu, 2020].

After running a series of iterations of hyperparameter tuning, the final hyperparameter values are chosen based on their importance in relation to the metric which is tracked. The focus will be on the collection of hyperparameters, which displays the highest reward. The hyperparameters will be changed based on the perceived importance of the parameter, taking into account if it has a positive or negative correlation to the score.

4.5.2 Action Masking

Action masking is a method to decrease the number of actions which an agent has to take into consideration. Thus, it can potentially accelerate the PPO model's learning. The implementation of the action masking method is inspired by a previous submission provided by Flatland [AIcrowd, 2021a]. The Flatland action-space consists of five distinct

actions: No-op, forward, left, right and stop. However cf. Section 2.1, only a subset of the actions are available at a given time step. Thus, the method excludes the actions which are non-available at a given state. E.g. if an agent is positioned at a node in the environment, where only the forward and the right action is allowed, the other actions are excluded. Furthermore, is the no-op action completely removed, since it is redundant as the other actions already allows the agent to continue with the same action. To test how action-masking improves the rate of learning, a comparative test is conducted in Section 5.

4.5.3 Frame Skipping

The general idea behind frame skipping is to skip cells between intersections. The implementation is as well inspired by a previous submission provided by Flatland [AIcrowd, 2021a]. The intuition behind the method is that paths between intersections only contains cells with redundant choices of actions. If an agent has to stop before an intersection, for example if another agent is crossing its path, it is only the cell right before the intersection where an agent has to take an active choice of stopping. Thus, all the cells which are not located right before an intersection is skipped. This ultimately means that an agent has fewer cells to learn the right course of action, which minimize the overall learning time. The performance is tested in Section 5.

4.6 Rewards

In regard to the reward function, the default reward signal provided by Flatland will be used (See Section 2.2). The reward function is focused on a global aspect, where rewards are given based on the agent's compliance with the time schedule. An opportunity lies to further improve upon the agent's learning, by introducing a more dense reward function where intermediate rewards are utilized, this is elaborated further upon in the discussion 6. The reason for not including a new reward function, is that it was chosen to focus on improving learning, by optimizing the PPO model with the inclusion of GAE and hyperparameter tuning.

Introducing a new reward function is a time-consuming task, as it involves a lot of trial and error; the agents might discover undesirable or even dangerous ways to obtain rewards [Sutton and Barto, 2018, p. 469]. A change in reward with good intentions could end up leading to a perverse incentive.

The same logic can be applied to the Flatland case where a reward for a specific behavior, e.g. avoiding deadlocks, could result in the trains never taking off, as this would be the easiest way to avoid a deadlock. In other words, the reward has changed the focus of the agent to solving a minor problem, to the detriment of the larger goal, arriving on time. This would be very time-consuming to change and test, and could be a project in itself

Performance Evaluation 5

The following chapter concerns itself with the evaluation of the RL-algorithm, PPO, the customized observation model and hyperparameter tuning. First, the ways in which the TMS agent will be evaluated is established. Subsequently, different training methods are outlined and investigated. In this regard, curriculum learning is elaborated upon. Ensuingly, the evaluation process and its results are presented 5.3. Finally, the final solution is optimized using different optimization methods, including a final hyperparameter optimization.

5.1 Evaluation Method

As mentioned in Section 1.2, a TMS has to be capable of generalization, be scalable and adaptable, in order to be considered applicable to a real world setting. As such, the developed TMS is evaluated on these three qualities, which leads to the following three performance metrics:

- **Normalized reward:** It is calculated by the following equitation [AIcrowd, 2020]:

$$Normalized\ reward = \frac{cumulative\ rewards}{max\ steps\ in\ an\ episode\ \cdot\ number\ of\ agents}$$

  Essentially the normalized return, normalizes the average received rewards in an episode, to get values in the range of $[-1.0, 0.0]$. This means that the highest possible value is 0, which is achieved if all agents reach their target location on time. The lowest score is -1.0, which are returned if none of the agents reached their target location, and they are the max time steps away from their target [AIcrowd, 2020]. Normalized rewards is the most essential metric, as it enables the evaluation of how the developed solution performs relative to one another. If the developed solution does not consistently get an adequate normalized score, it essentially means that the solution is not applicable as a TMS for Flatland. For instance, if a solutions' normalized score vary greatly for different environment setups, it indicates that the solution is not generalizable.

- **Completion rate:** The rate of agents which have reached their target. The completion rate is set to values in the range of [0, 1], where 1 correspond to 100% of the agents having reached their target location. Completion rate has been added as an additional metric to normalized rewards, as it enables to directly see the percentage of agents which have reached their target location.

- **Deadlock rate:** Has been added, to better understand the performance of a solution, as this is a central concern in VRSP, when railway network traffic needs to be rescheduled, either due to breakdowns or deadlocks. Two agents being deadlocked, ultimately means that they will not reach their target. This is also why it has been a focal point in the new implemented features in Section 4.4.3. If a solution has a high percentage number of deadlocks, it can indicate the methods used for enhancing coordination and communication between agents are insufficient.

5.1.1 Environment Configuration

In order to evaluate the developed TMS in its generalization capability and scalability, its performance has to evaluated on different environments with scaling complexity. Flatland provides an array of different variables, which can be used to configure the complexity of an environment. Of these, the three variables has been chosen as a means to configure complexity:

- **Number of agents:** Enables configuration of environments with different number of agents. Increasing the number of agents will result in a more complex environment, since more agents have to navigate the same space. For instance, it increases the probability of agents being in a future deadlock state or malfunctioned agents.

- **Width and height of environment:** These two combined variables can be used to alter the complexity in different ways. A smaller environment can lead to a more dense environment, if it contains a high number of agents On the contrary, a large environment results in the agent having to take more actions before it reaches its target destination. This can, as an example, increase the probability of agents not reaching its target location on time.

- **Number of cities:** Is included to mimic a classic railway structure, where the cities typically are populated with a high density of railway tracks, and the cities are sparsely connected [AIcrowd, 2021d]. Thus, increasing the number of cities results in more densely populated railway tracks, which increases the complexity of the environment.

Besides the stated variables which are used to configure the environments, it is worth mentioning that Flatland provides additional variables to configure the complexity of the environment. It includes variables such as the speed of the agents, malfunction rate, number of goal states, and a sparse rail generator. The sparse rail generator contains different variables, that for instance can change the number of parallel rails in the cities, or the number of rails between the different cities [AIcrowd, 2021d]. However, to limit the number of parameters to include when setting the configurations of the environments for training, the above-stated variable is set to the default values used in Flatland's test setup. The reasoning is that there are too many factors to account for when configuring the environments for training and testing. This is not feasible given the project's available resources. Furthermore, the chosen variables already enable many different instances of environments, with a large set of possible complexity configurations

5.1.2 Learning, Validation and Testing

Before testing different agent configurations and comparing them, all of them need to have their policy trained and validated first. This means that a framework for training, validating and testing configurations will have to be developed, so that they can be compared.

Curriculum Learning

Curriculum learning follows the intuition of how humans learn, by introducing basics first, and then steadily adding more difficult concepts. In principle, curriculum learning is about dividing a training set into tasks, from now on referred to as levels, ordered from easy to difficult [Narvekar et al., 2020, p. 1]. A level consists of an environment configuration, with parameters such as dimensions and number of agents. Each of the levels is run for a predetermined number of episodes.

There are, however, many ways to iterate through a training set from easy to difficult, by ordering the way in which the agent is presented to the different levels of the training set. The specific ordering is defined as sequencing, from now on denoted as ordering methods [Narvekar et al., 2020, p. 9]. An example of an ordering method could be to iterate through all levels from a training set 200 times, until the end of the set is reached.

The aim of curriculum learning is to transfer knowledge between levels, leveraging what the agent has learned in one level to the next [Narvekar et al., 2020, p. 1]. This is to prevent the agent from initially having to interact with an environment that is too complex too early, where it would rarely get a reward, resulting in slow learning [Narvekar et al., 2020, p. 1-3]. This should, in theory, be mitigated by the curriculum learning method, resulting in both improved and faster learning.

Traditionally, training sets and ordering methods are manually defined for each problem, rather than using automatic methods [Narvekar et al., 2020, p. 3-4].

Validation

In order to identify when the PPO model has been trained adequately, it is validated regularly with an interval of n episodes. As such, its rate of learning can be tracked over time, after which training is stopped, once the learning rate has stagnated. The training process can either be stopped with an automatic stopping mechanism, or it can be done manually. A manual visual inspection is used in favor of its automatic counterpart, due to unreliability issues which could arise when using an automatic stopping mechanism. An example of training convergence is shown in Figure 5.1.

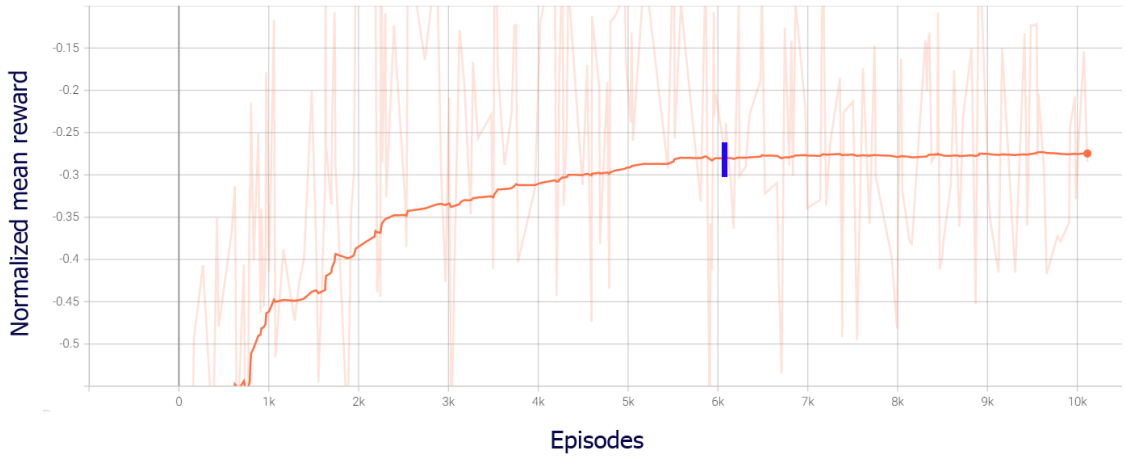


Figure 5.1: In the graph, the point of convergence is illustrated by the blue line. This is the point where it is assessed to converge, as the rise in normalized mean rewards stagnates.

Furthermore, an Exponential Moving Average (EMA) is used to filter out noise and identify the overall trend in the graphs. In short, the EMA reduces noise by setting a data point value according to an average of previous data points over a defined period. In this regard, a weight which decays exponentially is applied to each data point, resulting in greater emphasis on recent data point [Ang, 2021, P. 40]. This enables to smooth out outlier data points, and better track trends over time. This is also visualized in Figure 5.1, where the line with lower opacity is the unprocessed data points without smoothing. As evident, the unprocessed data points do not provide a sufficient overview of the general trend of learning, in order to determine when the rate of learning converges. This has led to the choice of using smoothing an all graphs from this point on, as two different plots are very difficult to compare without.

Test

After an agent configuration has been validated to be near its optimal policy, it is tested on a local instance of the official Flatland testing utilized. The testing levels and their parameters can be seen in Table 5.1 [AIcrowd, 2021b]. The levels are configured to have more frequent malfunctions as well as more speed profiles as they progress. This correlates to what is sought to be tested in regard to adaptability of the agents. Additionally, to take notice of in the Flatland test setup, is that for each level, 10 environment instances are run. If the average percentage of done agents is less than 25% within a level as showcased in Figure B.1 and Figure B.2, the test does not proceed to the next level, as it is the minimum requirement for passing a level.

Level	Agents	x-dimension	y-dimension	Number of cities	Environments
1	7	30	30	2	10
2	10	30	30	2	10
3	20	30	30	3	10
4	50	30	35	3	10
5	80	35	30	5	10
6	80	45	35	7	10
7	80	40	60	9	10
8	80	60	40	13	10
9	80	60	60	17	10
10	100	80	120	21	10
11	100	100	80	25	10
12	200	100	100	29	10
13	200	150	150	33	10
14	400	150	150	37	10
15	425	158	158	41	10

Table 5.1: Official Flatland test set. It showcases the number of training levels used, the variables used to configure the complexity of the environments, and the number of environment instances for each level

5.1.3 Curriculum Design

Based on the previously described idea behind curriculum learning, a specific training set to be used for training the agents is constructed in Table 5.2.

Level	Agents	x-dimension	y-dimension	Number of cities
1	7	30	30	2
2	10	30	30	2
3	20	30	30	3
4	50	30	35	3
5	80	35	30	5

Table 5.2: The training set consist of five levels, where each level increases in complexity by the use of the configurable environment variables as described in Table 5.1.1. The setup of the levels follow the structure used by Flatland's test setup (See Table 5.1).

The choice of including only five levels, is due to the time constraints, as the increase in the difficulty of a level rapidly increases running time. To validate when the performance of a training session converges, a validation interval of 200 episodes is chosen. The validation setup is fixed to five levels, which mimics the first five levels of the Flatland test, where each level is evaluated one time. In regard to the returned normalized rewards, after a validation run, the mean of the five evaluated levels is refereed to as the normalized mean reward.

The environments, used in both training and validation, are randomly seeded, meaning that new environments are generated each time with the same environment configuration. This

choice is based on the fact that a solution needs to be tested in regard to its generalizability, and using the same environment seeds, could result in a risk of over-fitting. However, when testing, the same seed will be used for the environment instantiations, because it gives a single point of reference when comparing two solutions' performance.

Besides the construction of levels, the order in which the levels are evaluated has to be considered as well, as a specific ordering can greatly improve the rate of learning for the PPO model. To decide the ordering of levels, two ordering methods have been constructed – incremental and sequential – which have been depicted in Table 5.3.

Sequential	**Incremental**
1	1
2	1
3	1
1	2
2	2
3	2
1	3
2	3
3	3

Table 5.3: Demonstration of the ordering methods, Sequential and incremental, where a training set of three levels is given, and assuming that the rate of learning have converged after nine environments. Each number represents the level in which one environment is run

The Incremental Ordering Method

The idea behind the incremental ordering method, is that all levels in the training set are ordered by difficulty, and in each level the PPO model is trained until the learning curve converges. When this happens, the trained policy is transferred to the next level, and so on until all levels have been completed. This is depicted on the right in Table 5.3, where level 1 is run until convergence, after which the next level is initiated and so on.

An advantage of the incremental approach is that it follows the curriculum learning formula of ordering levels from easy to difficult. As such, one can with greater certainty be sure that the knowledge gained in one level is going to be transferable to the next. However, a disadvantage is that the agent near the end of the training set only interacts with larger-scale environments and no smaller environments from the previous levels. A seeming shortcoming of the incremental approach could be that be the agent forgets how to solve smaller levels, as they are the first levels that the agents are exposed to during training.

The Sequential Ordering Method

The sequential approach, which is depicted on the left side of Table 5.3, loops over all three levels iteratively until the normalized reward of the whole training set converges. A potential advantage of this approach is that it runs through each level without over-saturating its learning to a single level. The intuition is that when running through the training set sequentially, the agent will remember all level sizes as it is constantly being

presented with varying sizes while training, and will therefore potentially be better at tackling smaller levels than the incremental approach. However, a downside with the sequential method is that the agent configuration has not yet converged on a previous level, when the knowledge is transferred to the next, which could make the knowledge transfer less effective. Which means there is a risk that training will be slower, as the agent very quickly gets introduced to levels of high complexity.

Summarization

The training set consists of five levels, each having a different environment complexity configuration. The goal of curriculum learning is to enhance the learning of the agents, in regard to minimizing learning time, and to improve the final test result of a solution. Furthermore, three ordering methods have been constructed to further improve upon the learning of agents. The selection of an ordering method, in combination with the five levels of the training set, is based on the following factors:

- Fastest rate of convergence in the validation of agents training
- The Highest average received rewards in the Flatland test setup

The method, which performs the best, will be used as the training setup. To combat the stochastic nature of the environment configuration and the agents policies, two iterations will be sought to be conducted for each training, validation and test. Ideally, more iterations should be used to combat the variance between results, however due to limited constraints, two has been chosen. To name a few factors which results in variance between testing; the randomly seeded environments, the different fractional speed of agents, the random generated malfunctioning agents and the agent's policy.

5.2 Evaluation of Components

Having established the evaluation method, the evaluation of the different components in a final solution needs to be accounted for. This involves evaluating the curriculum learning method, the newly implemented features in the tree observation method, the performance optimization methods (frame skipping, action masking), and the tuning of the hyperparameters in the PPO model. The goal is to evaluate the different configurations of the components, in order to choose the configurations which result in the highest performance compared to the baseline configuration.

Feature Evaluation

As accounted for in Section 4.4.2, the default tree observation method provides 12 features, where six additional features have been implemented in a new observation method. Ideally, all combinations of features should be tested, in order to choose the combination of features that provides the best performance based on the evaluation metrics established in Section 5.1. However, this is not feasible due to time constraints. If all combinations were tested, it would result in $2^{18} = 262,144$ feature combinations.

To make the testing of features more feasible, only two feature combinations are tested:

1. The original tree observation with 12 features
2. The new tree observation with a total of 18 features

The drawback of this approach is that it is not guaranteed to find the optimal combination of features, as there is no evaluation on how some features may negatively impact the performance of others. However, this approach provides a rough estimation of the benefit of implementing the new features.

Three Depth Evaluation

The tree depth of the original tree observation is two. As mentioned in Section 4.4.2, the time complexity of the tree exploration method is $\theta(2^n)$, where n is the three depth. The tree depth should therefore be big enough to provide the agents with the necessary information in order to reason in the environment, but small enough to still be feasible in relation to the time performance. The agents need to adhere to the following time constraints in Flatland challenge 3 [AIcrowd, 2021f]:

1. A maximum of 10 minutes of initial planning time
2. A maximum of 10 seconds per time step to choose agent's next actions
3. A maximum of 2 hours for the full evaluation

Furthermore, there is also a time constraint in this project, so the training time is also limited. Due to these constraints, different tree depths are not tested, as it is deemed unlikely that an increase or decrease in three depth would improve the overall solution:

- A tree depth of one is judged to be too small, as the agents would only be able to observe the information on their current path. It would therefore be very difficult to reason in the environment and calculate which path should be chosen.

- If the tree depth was increased to three, the risk of breaking the time constraints would be too high.

Therefore, the final solution could potentially be improved in the future if the solution was optimized in regard to the time metric, as this could allow for an increase in the three depth while still adhering to the constraints.

Frame Skipping and Action Masking

Frame skipping and action masking are both evaluated individually and combined. The setup, which provides the biggest performance gain, will be included in the final solution. As described in Section 4.5, the theoretical improvement should be faster learning. Thus, two criteria are important when evaluating frame skipping and action masking: Convergence speed during training and the actual performance difference in the final test.

5.2.1 Evaluation Overview

The approach for the evaluation is shown in Figure 5.2. In summary, the approach is choosing the curriculum learning method firstly, then the observation, and then optimized with frame skipping and action masking, and finally hyperparameter tuning. The ending result is the final solution.

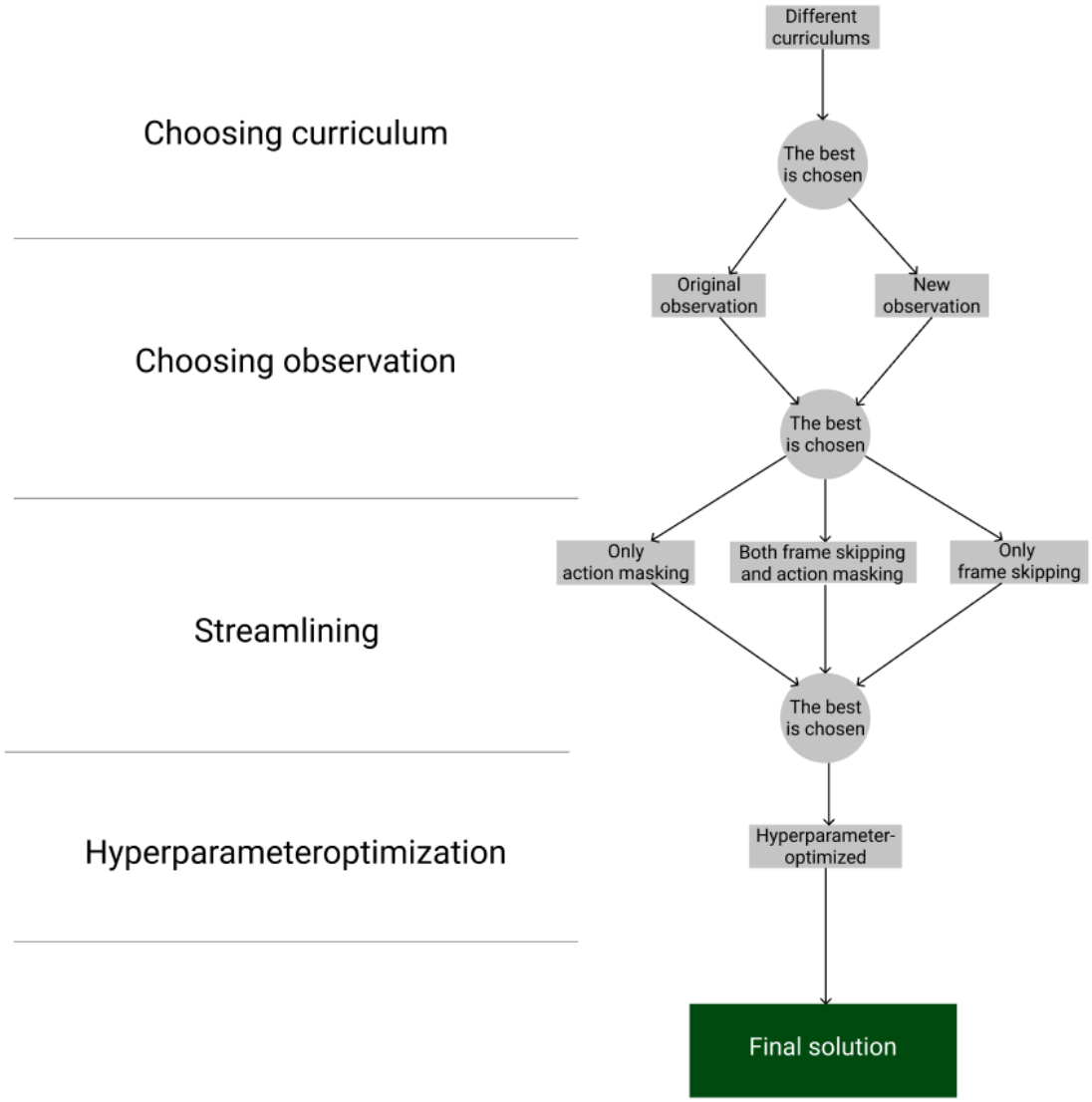


Figure 5.2: Overview of the evaluation approach. The squares in the model correspond to the test's going to be conducted. The circles correspond to the points where it is decided which method is included in the final solution

5.3 Execution of the evaluation

Having constructed an evaluation method, the foundation is laid to proceed with the testing of the developed solution. Following the structure of the evaluation approach outlined in Figure 5.2, the first component to test, is the curriculum setup.
However, in the testing phase, it was found that the testing data didn't seem to correlate with the validation data. This changed how the results will be evaluated and is therefore clarified firstly.

Flatland Test Setup

As described in Section 5.1.2, Flatland's test setup has been utilized. However, during evaluation, this specific test setup have proven to be ill-fitted for the evaluation purposes of this project, as it unfortunately provides inadequate and unreliable results. In particular, the following two attributes of the test setup has been assessed to give arise to difficulties:

- *High variance*: The test setup makes use of 10 episodes per level. This amount of episodes has, however, been identified as insufficient. Due to the stochastic nature of training, there is a relatively high variance on the test performance of all solution. As such, more than 10 episodes are needed in order to deduce anything certain from the test.

- *25 % rule:* In order to proceed to the next level, it is required that 25 % of the agents have reached their target in the current level. otherwise, testing is stopped. This stopping mechanism has proven to be inconvenient when taking the high variance into account. With high variance, it may be up to chance whether a solution is able to proceed to another level. This leads to an unfortunate bias towards solutions, which may have managed to proceed to a higher level, despite it being a result of pure coincidence.

In light of the above, the Flatland test-setup cannot be used to conclude anything with certainty, but can only be used in an indicative manner. Due to these adversities, the training and validation results have been deemed to carry more weight. As such, their results are used as a fallback to draw performance conclusions with. An example of the Flatland test-setup providing unreliable results is shown in Figure 5.3.

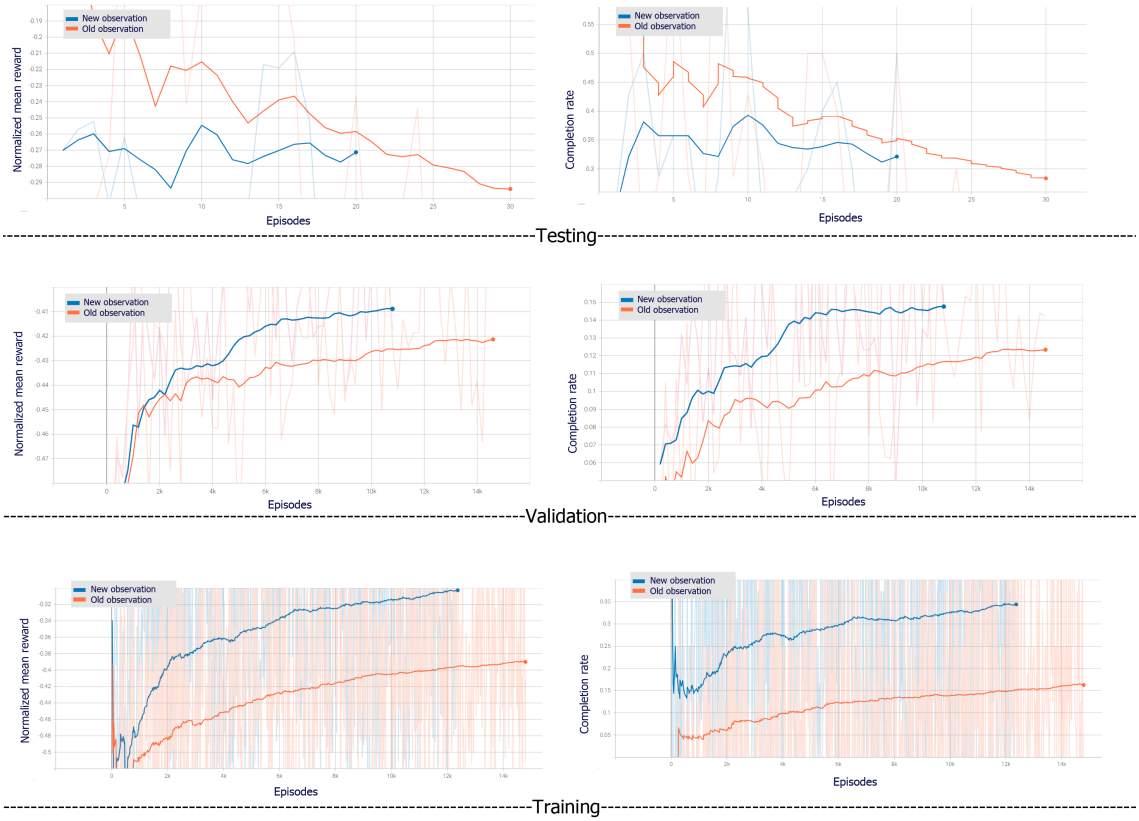


Figure 5.3: Overview of the test, validation, and training scores for the new observation (blue) vs. the old observation (orange).

The Figure shows that the training and validating data suggests that the new observation outperforms the old observation. However, the test results suggest that the old observation outperforms the new observation.

5.3.1 First Curriculum Evaluation

In proceedings of the training setup described in Section 5.1.3, the two ordering methods (Sequential and Incremental) have been tested in combination with the five training levels. The results from the test revealed some shortcomings, both in regard to the training set itself and the ordering methods.

It was found that both the initial and the subsequent levels in the training set were too difficult, resulting in a slow rate of learning. For example, the incremental method, where 6.5 hours was spent on the first level alone, before it converged (see Figure A.17). A further observation from the first iteration of the curriculum method, was that the transfer learning between levels was limited, when using the sequential ordering method. This was concluded as the sequential method trained for an entire day, without showing any sign of converging, and that the completion rate of agents in the higher levels in the training set was almost non-existent (see figure A.17). Taking these findings into consideration, it was not feasible to continue with the initial training set, and a new training set has to be constructed. Furthermore, in the incremental approach's current state, it requires manual inspection and work before proceeding to the next level. This makes it a tedious

and seemingly error-prone approach when several configurations have to be trained. An automatic stopping mechanism is a viable solution to this problem, but may prove to be unreliable, due to having to consider every possible edge-case. Thus, the incremental ordering method should be replaced with an ordering method requiring less manual labor.

5.3.2 Second Curriculum Evaluation

Based on the results of the first iteration, and the shortcomings hereof, some changes were needed. Firstly, a training set with an initial smaller amount of agents, as well as smaller difficulty jumps between levels, has been devised – see Table 5.4. This is to improve transfer learning, i.e. knowledge gained in a level is transferable to the next [Narvekar et al., 2020, p. 13]. Preferably, the reconstructed training set should be tested, in order to establish the appropriate difficulty spacing between levels, as this potentially could improve the transfer of learning between levels. This could result in the training reaching the rate of convergence significantly faster. However, this has been excluded from the project, as the spacing between difficulty levels can be configured in multiple ways, which would be too comprehensive.

Level	**Agents**	**x-dimension**	**y-dimension**	**Number of cities**
1	2	30	30	2
2	4	30	30	2
3	8	30	30	2
4	12	30	30	3
5	16	30	30	3

Table 5.4: Second iteration of the training set

The training set remains at five levels in this iteration, but with decreased difficulty overall. The values of the x- and y-dimensions, are fixed as these parameters are only scaled in the test set, when 50 or more agents are introduced. So to follow the test set, environment size is not going to be increased as the new training set has a maximum of 16 agents. So instead of evaluating scalability by changing the environment size, it is going to be evaluated by increasing the density of agents and cities in the environment.

Since the incremental ordering method was discarded in the last iteration of testing, a new ordering method is introduced, the progressive ordering method, which can be seen on the right in Table 5.5.

Sequential	**Incremental**	**Progressive**
1	1	{1} × n
2	1	{1,2} × n
3	1	{1,2,3} × n
1	2	1
2	2	2
3	2	3
1	3	1
2	3	2
3	3	3

Table 5.5: Demonstration of the ordering methods sequential, incremental and progressive, where a training set of three levels is given.

The progressive approach is very similar to the sequential method, but implements some aspects of the incremental approach to incentivize transfer learning. However, the progressive method adds some initial training before ending up the same as the sequential method. As shown in Table 5.5, it runs level 1, then level 1 and 2, then level 1, 2 and 3. All of these are run for n time steps until it reaches the max level of the training set, from which point it becomes identical to the sequential method, looping over all levels until the learning rate of the whole training set converges.

This approach has been implemented, because of an observed problem in the first iteration of curriculum testing using the sequential method. The problem was that the agent configurations had difficulties improving their policy, as they were introduced to levels that were too difficult too early. The logic behind doing the initial steps in the progressive method, is to better prepare the agent to handle the larger levels, instead of introducing the most difficult levels of the training set as soon as the first run.

This means that the progressive method could be seen as a hybrid of the incremental and sequential method. Similar to the incremental method, it has a larger degree of experience when tackling harder levels, and similar to the sequential method it has all difficulty levels in memory when training ceases.

Results of Second Curriculum Evaluation

The validation results of the new curriculum with the progressive and sequential ordering methods got the results shown in Figure 5.4.

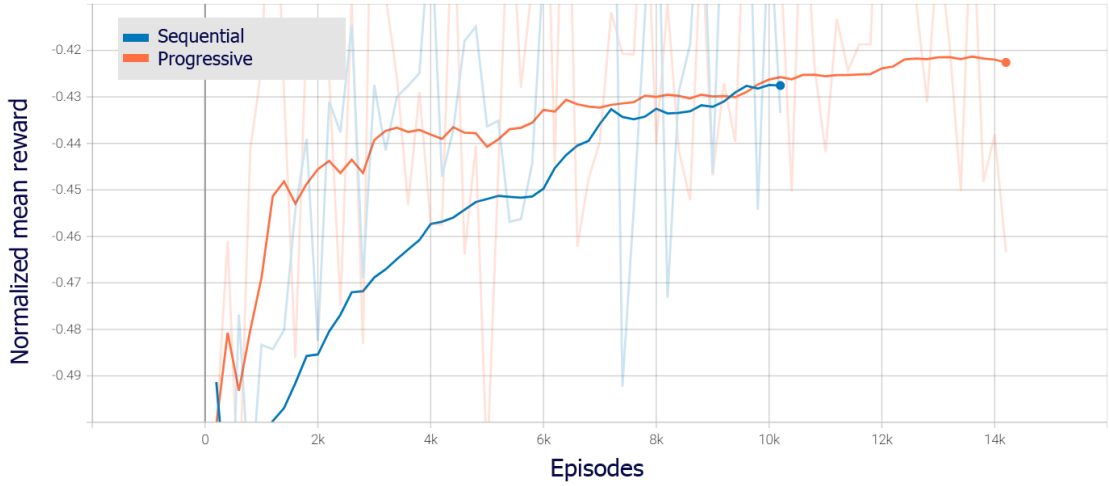


Figure 5.4: Validation results of the Sequential ordering method (blue) and progressive ordering method (orange) with the new training set.

Figure 5.4 shows that the normalized mean reward of the sequential and incremental ordering methods at 10,000 episodes was approximately the same. This is expected as progressive ends up with the same ordering as sequential. However, the rate of learning for the progressive ordering method outperformed the sequential ordering method.

Given these two results, the choice has fallen on using the progressive ordering method, which will be the default ordering method for all further training.

5.3.3 Observation Testing

As mentioned in Section 5.2, only two feature combinations are tested: The original tree observation and the new tree observation with six additional features. The original tree observation was used to establish the baseline, so the result of this has already been concluded in Section 5.3.2. Therefore, only the new observation needs to be tested in order to be able to compare the two. The result of this test is shown in Figure 5.5, where the previous test of the original tree observation is also illustrated.

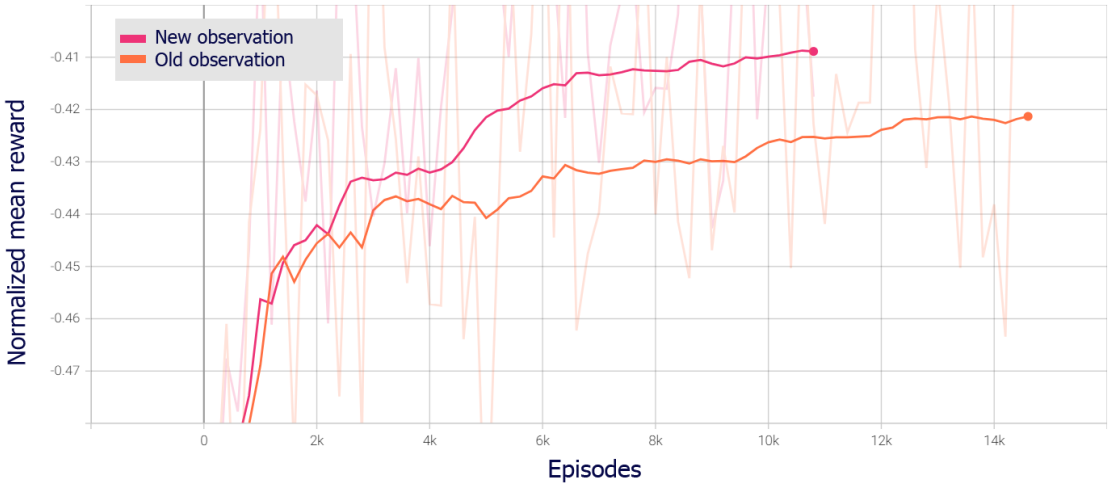


Figure 5.5: Validation results showing the Normalized mean reward for the new observation vs. the old observation

The two curves in Figure 5.5 suggests that the new tree observation is better than the old, as normalized mean reward is better throughout almost all the episodes. Both observation-curves have converged in the Figure, where the new observation scores approximately -0.41 and the old observation scores approximately -0.42. The score is therefore increased by 0.01, which is circa 2.4 percent. However, it is also important to take the overall completion rate into account. The completion rate is shown in Figure 5.6.

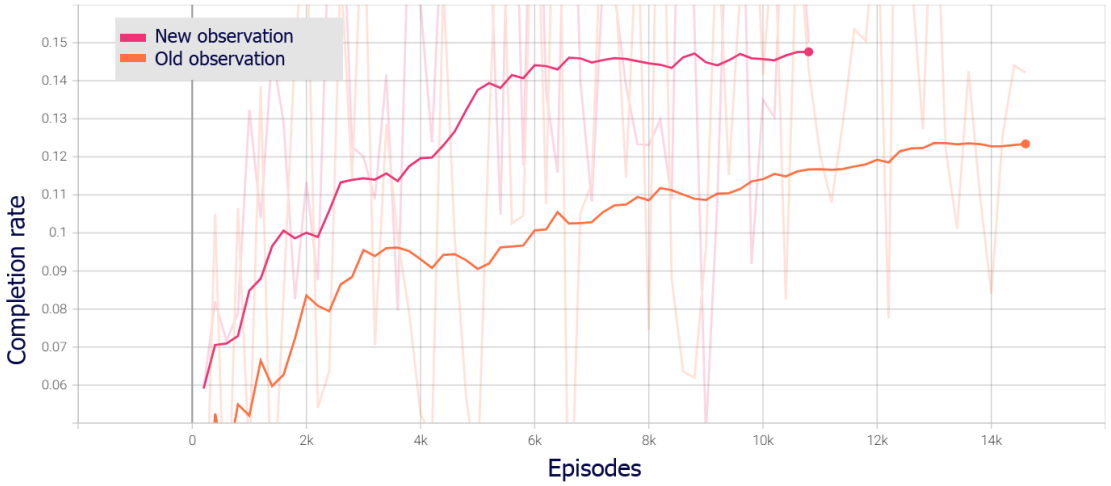


Figure 5.6: Validation results showing the completion rate for the old observation method vs. the new observation method

The Figure suggests that the new observation outperforms the old observation in terms of the completion rate. The end value of the new observation is approximate 0.15, and the end value of the old observation is approximate 0.125. Therefore, the completion rate is increased by 20 percent. The Figure also shows that both curves converge at approximately the same time, which suggests that the new features does not impair the rate of learning.

The new tree observation is therefore chosen as the observation method that will be used going forward, as it outperforms the old observation in regard to both the normalized mean reward and the completion rate.

It should, however, be noted that the initial plan was to include a graph, showing the difference in deadlock percentage, as the new observation had features that were implemented with the intention of decreasing the number of deadlocks. However, as mentioned in Section 5.3, the testing results were too unreliable, and unfortunately the validation data did not include the number of deadlocks. For these reasons, the number of deadlocks cannot be directly compared for the two observations.

5.3.4 Testing of Frame Skipping and Action Masking

As of now, it has been concluded that the final solution will be utilizing the progressive ordering method as well as the new tree observation. The next step is to test the implementation of frame skipping and action masking on the new included components. The test which will be conducted are respectively action masking, frame skipping and the two in combination as stated earlier in Section 5.2. They will be compared to the previous result from the new included tree observation, where the best performing combination will be chosen. The results from the test are visualized in Figure 5.7.

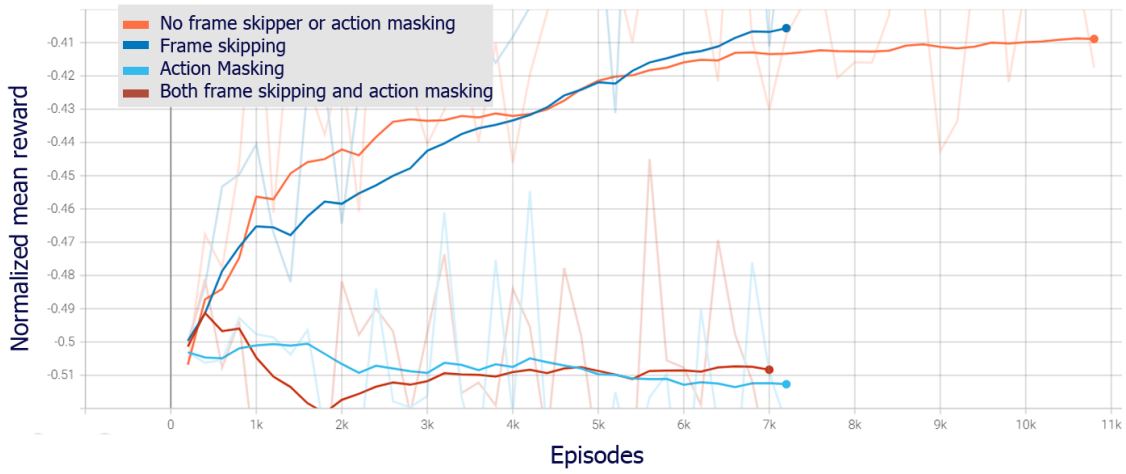


Figure 5.7: Validation results from validating frame skipping and action masking combinations

As evident from the Figure, action masking and action masking combined with frame skipping both scores considerable lower than the new observation and frame skipping, with a normalized mean reward around -0.51. A further examination of their normalized mean rewards, showcases that the normalized mean reward does not improve when the number of episodes increases, but actually stabilizes at a lower level. This indicates an error in the implementation of action masking, as there is a discrepancy between the actual implementation of action masking, and what the theory describe as mentioned in 4.5.2. As a result, action-masking and action masking combined with frame-skipping are ruled out based on the discussed results.

Examining frame skipping and the new observation method in Figure 5.7, the new observation model initially has a higher rate of learning from around 1000 to 4000 episodes, however the point of convergence are for both roughly estimated around 7000 episodes. According to Section 4.5.3, Frame skipping should result in a faster convergence, which is not the case. Looking at the normalized mean reward around the point of convergence, frame skipping has a value of approximately around -0.405 compared to the new observation value of approximately -0.413. This results in a performance difference of 0.008, which is a difference of circa 2 percent. Comparing the percentage in completion, the same trend is applicable, which is visualized in Figure 5.8 below. At the point of convergence, frame skipping has a value of approximately 0.16, compared to new observation with a value of 0.15, which means that frame skipping has approximately 7 % higher completion rate.

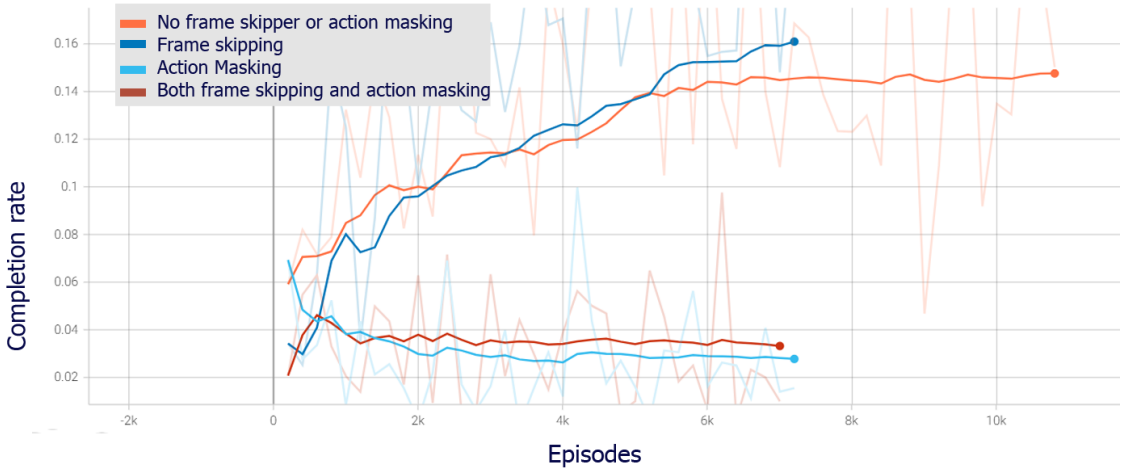


Figure 5.8: Validation results showing completion rates for frame skipping and action masking combinations

In conclusion, the new observation combined with frame skipping performs better at the point of convergence, but does not result in a faster convergence, as the initial expectation was. However, the small improvement in final performance from the training of the agent's, still means that the addition of frame-skipping adds value to the solution and is included going forward.

5.3.5 Hyperparameter Tuning

As both the optimal curriculum, observation and additional methods have been chosen, the hyperparameter tuning search space from Section 4.5.1 is now applied in an effort to maximize the results. Hyperparameter tuning was run 100 times using the progressive ordering method on the new observation model with frame skipping enabled, as this combination have proven to result in the highest score.

In an effort to generate as many hyperparameter runs, as possible, the amount of episodes in a hyperparameter run will be limited to 2500 episodes. This hard-cap is picked based on the generally observed convergence, at around 2500 episodes, as depicted in Figure

5.9. As the training earlier has been found to have a high correlation to the validation score, the normalized reward in the training set is used as the metric to optimize towards. The training set is used, as it provides data on each episode, instead of the validation which runs every 200 episodes. This means that using validation data, only would produce approximately 12 data-points against 2500, when using the training data.

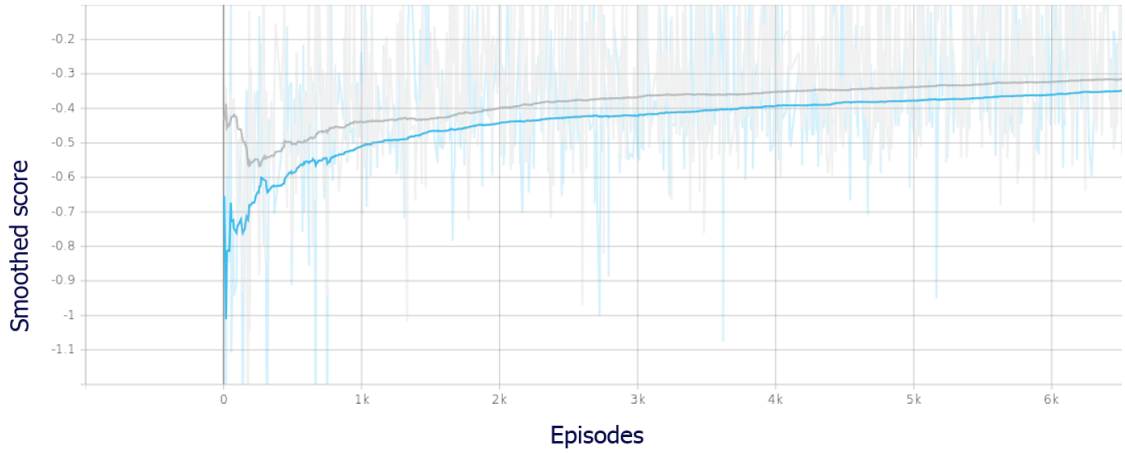


Figure 5.9: Graph displaying that the average convergence point of the aforementioned training configuration is around 2500 episodes

In order to optimize the rating of the resulting hyperparameter runs, a smoothed normalized mean reward will be used, such that the rating of a score is not determined by a temporary upswing in performance, but its average score throughout training. Finally, it should be noted that the pitfalls of using a shorter amount of episodes for hyperparameter training. could be that the hyperparameter sweep would only prioritize the first 2500 episodes, and afterwards plummet to 0.

After the 100 hyperparameter runs were done, the general trends and likenesses of the top configurations were identified, by filtering the top sweeps according to their score. The enabled filter is an accumulation of the best runs performed throughout the hyperparameter sweep. Wandb allows for identifying trends among the top-runs. In Table 5.10 the *importance* correlates to the impact that the hyperparameter has on the model, while *correlation* showcases if a higher or lower value is beneficial.

The table indicates that a larger hidden size is beneficial, suggesting that the maximum hidden size before diminishing returns have not been met yet. The amount of epochs is also shown to be beneficial, which is expected due to the agent being able to iterate over its training data more often. The learning rate is very individual from model to model, but showcases that a higher rater than lower learning rate is beneficial. The learning rate would ideally be further tuned later on, as this specific parameter has a high impact on model performance. Lastly, the table shows a positive impact on a higher batch size and discount factor, which is expected. The most surprising finding is that a larger buffer size have a negative impact on the performance, as one would naturally assume that more data would make for better decisions.

Figure 5.10: Table showcasing the correlation between importance of hyperparameters in regard to the model and environment. As well as the negative and positive impact of a high and low setting for each hyperparameter

After having run 2500 episodes training for 100 sweeps, 10,000 episodes of training were run on the top eight sweeps and afterwards the three best configurations were picked, as they showcased very promising results. A comparison of the previously tested progressive ordering method with the new observation model and frame skipping can be seen depicted in Figure 5.11, which showcases the large impact of hyperparameter tuning. The Figure displays validation scores, where the hyperparameter tuning were done on the training score. which supports the hypothesis of transferability of performance between the validation and training scores.

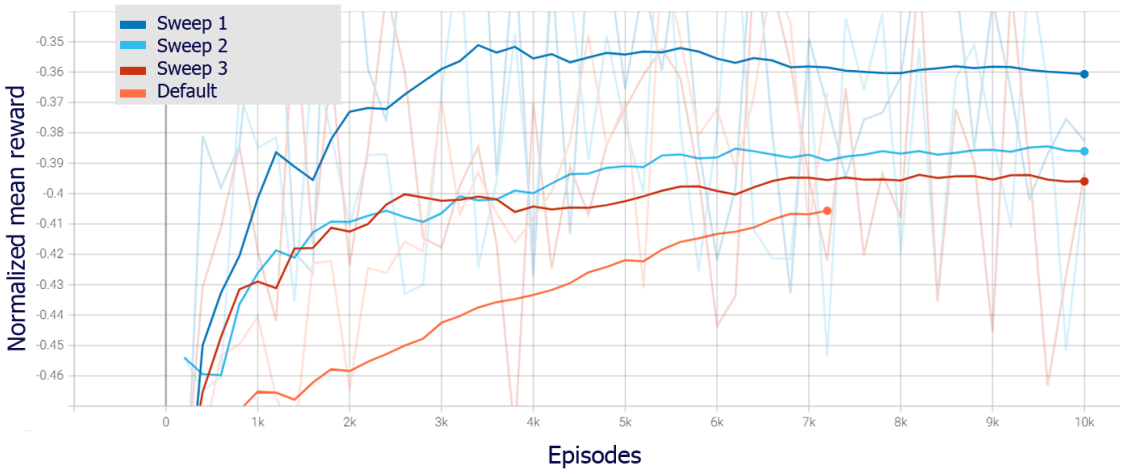


Figure 5.11: Graph displaying the validation results of the top 3 hyperparameter runs, as well as the progressive ordering method with the new observation and frame skipping enabled.

In the Figure, it can be seen that all the top hyperparameter sweeps reaches faster convergence when looking at the normalized score for validation, where all sweeps are fully converge at 4000 episodes, the default (progressive ordering method, with new observation and frame skipping) only slightly seems to converge at around 7000 episodes. Additionally,

the final score for the sweeps are respectively: 0.396, 0.386 and 0.36. With especially being the ladder that surprises with an increase in score of approximately 13 % in comparison to the default curve.

In Table 5.6, the final solution is compared to a PPO model with default hyperparameters. Notably, the activation function has changed from the Tanh, used in the PPO paper, to ReLU. The buffer size has increased to 51,200. Thus, more experience is gathered before the PPO model makes a policy update. The mini-batch has increased to 256, meaning that more samples are used for a policy updates. The discount factor has been increased to 0.99, further prioritizing rewards in the future, instead of the immediate ones. The clip factor has been changed to 0.2, which also was the most prominent clip-factor in Schulman et al. [2017]. Somewhat surprisingly, the amount of epochs has been reduced to 5, while the model still outperforms the default one with 10 epochs. Lastly, the entropy coefficient has been set to 0, which in theory reduces the exploration in the environment.

Hyperparameter	**Default value**	**Final solution**
Activation function	Tanh	ReLU
Buffer Size	32,000	51,200
Mini-batch Size	128	256
Discount (γ)	0.97	0.99
GAE parameter (λ)	0.95	0.95
Learning rate (α)	$5 \cdot 10^{-5}$	$5 \cdot 10^{-5}$
Hidden layer size	128	128
Number of hidden layers	2	2
PPO clip factor(ϵ)	0.1	0.2
Epochs	10	5
Value function coefficient(c_1)	0.5	0.5
Entropy Coefficient(c_2)	0.01	0

Table 5.6: Table of hyperparameters with the default values and optimized values

5.3.6 Final Results

Figure 5.12 shows the test results from the default PPO model configuration, the added components and the final solution.

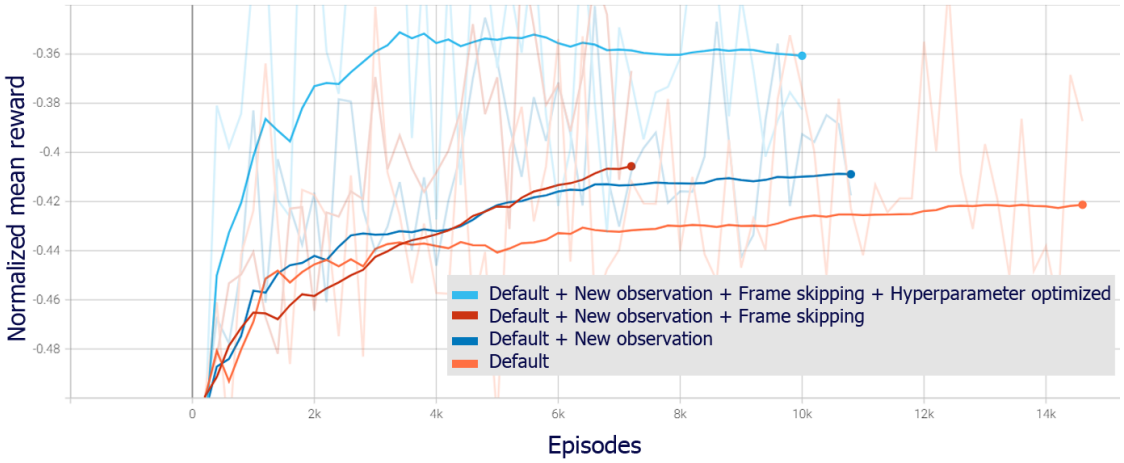


Figure 5.12: Graph depicting the validation results of the optimization of the model.

The Figure shows that the curve of the default PPO model has a normalized mean reward of approximately -0.42 at the point of convergence, compared to -0.36 on the curve of the final solution. This results in an overall improvement of approximately 14,3%, and indicates that the efforts conducted in the project to increase the scalability, generalizability, and adaptability of the TMS solution in Flatland's environment to some degree has been realized.

Recall, the purpose of implementing a new observation was partially to increase the adaptability of agents. However, since the testing of deadlocks was discarded along with the general testing, it is difficult to say anything conclusive about the adaptability of the final solution. Nonetheless, the created TMS still has indications of having better adaptability than the baseline solution. The reasoning behind this, is the fact that the new observation with additional features got a better normalized mean reward than the old observation in the validation data. The purpose of adding features was to improve the quality of the agents' decisions, in that features such as *has deadlocked agent* and *dispute next cell*, provides the agent with a more detailed view of the environment, giving it better conditions for making a good decision. In other words, the gained information of the new observation, could help the agent adapt to the changes in the environment, e.g. by potentially avoiding deadlocks, thus increasing adaptability.

Examining the final solution's scalability, the training data from the final solution is used to illustrate how the solution performs in regard to the five levels that it iterates over in the progressive curriculum, which is shown in figure 5.13.

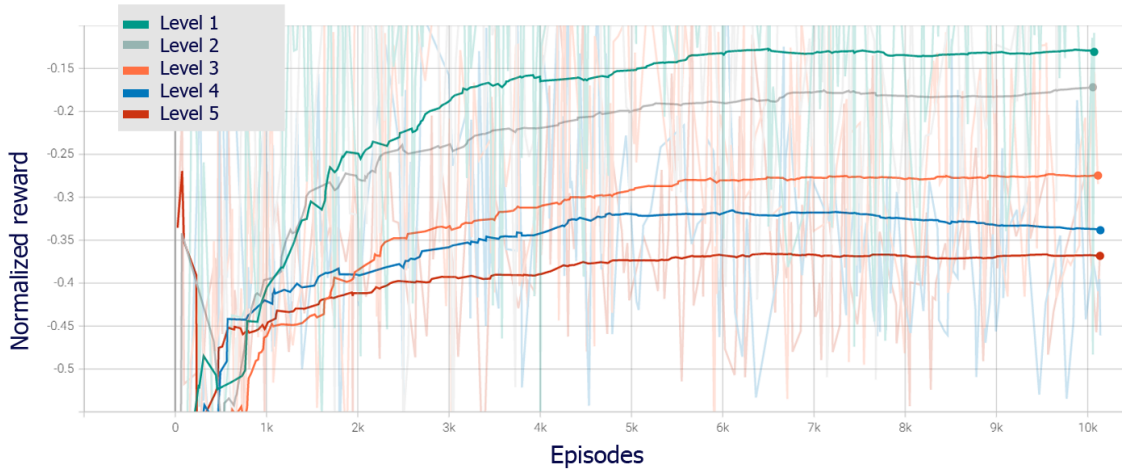


Figure 5.13: Graph displaying the training results for the five difficulty levels displayed, from the final solution's training data

Examining level 1 (2 agents, 2 cities, 30×30 grid) displayed in Figure 5.13, the normalized reward is circa -0.13 at the point of convergence. From the graph, it can be observed, that the normalized rewards decreases relative to the increase in difficulty by the levels. At level 5 (16 agents, 3 cities, 30×30 grid) the normalized mean reward is approximately -0.37. This is a decrease in returned normalized reward of 0.24 (182%). It is expected that the normalized reward decreases when the difficulty is scaled, but the significant difference in normalized rewards when the environment is scaled, suggest that improvements are necessary to make the solution more scalable. Especially when environments are scaled to even higher complexities.

The final solution scores relatively well on a considerably harder validation set, than what it has trained on, as can be seen in figure 5.14.

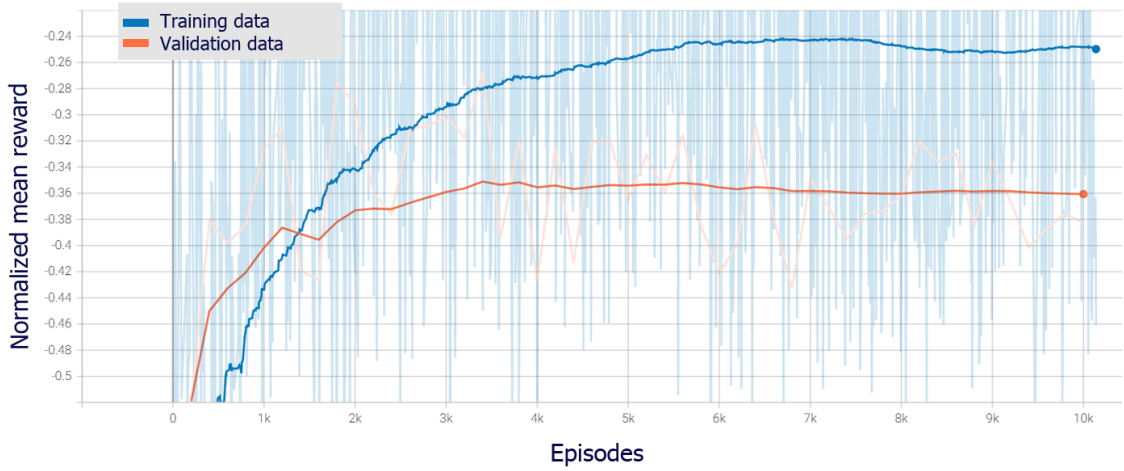


Figure 5.14: The training and validation data for the final solution

Recall, that the validation set levels follows the first five levels of the Flatland test set. In Figure 5.14, the blue curve, which is the normalized mean reward of the training set, maps relatively well to the normalized mean score of the validation set, as the trajectory of the curves are very similar. Given this, one can postulate that if the solution had trained on a larger training set, it could be able to map to an even harder validation set as well, so in this regard the solution is somewhat scalable.

The final solution was observed to have an overall generalizability, because the PPO model learns from randomly generated environments of varying complexity. However, a high variance in the training results for the PPO model between environments within the same level was observed, which indicates that the generalizability is limited. It should be noted that this has been observed on validation and training data, where the policy has not finished training yet. As such, in order to draw a meaningful conclusion on the generalizability of the solution, further investigation would be needed. For instance, one do an isolated test with a policy that has finished training, and stochastic variables such as speed profiles and malfunctions should be turned off to more easily observe trends.

Discussion 6

This chapter is devoted to reflect on four different key aspects of the developed TMS, namely the observation model, PPO model, reward optimization and performance evaluation. The reflections elaborate on potential improvements, which can enhance the proposed TMS.

6.1 Observation Model

The observation could be optimized by doing additional testing on different configurations. This includes increasing the tree depth and allowing for additional communication between the agents and trying to apply alternative data-structures to the challenge.

Tree Depth

The tree depth of the observation could be further experimented with. If the solution ran faster – either by improving the PPO agent or the observation method – then the tree depth could potentially be increased while still complying with the time limits, set by Flatland. This would allow the agents to have a greater overview of the different paths they can take, and as a result, might improve performance and increase the scalability of the solution. The intuition being that an agent is probably more likely to reason effectively in the environment, if they can predict potential conflicts at an earlier point in time. However, the positive effect of increasing the tree depth would probably be diminishing, as uncertainty increases in relation to how far forward the agent tries to predict.

Communication

If the agents were able to make better predictions in the environment, the performance and scalability could potentially be increased. This could, for example, be achieved by the agents communicating, at which time step they expect to be at each switch. This might make the merging of agents' paths easier and less error-prone. If, for example, two agents' paths merges, then the speed and excess time for the agents could be taken into account. Generally, the fastest agent should probably be at the front, as this would allow the agent to keep its maximum speed instead of decreasing its speed to the other agent. However, if the slowest agent has a low excess time, then it could also be more efficient to let that train merge first.

The communication could be facilitated by a global observation, e.g. by the use of a centralized critic. An optimal solution might include a hybrid solution of the tree and

global observation. Additionally, observations not provided by Flatland could be utilized, such as a graph observation or a mix of other data structures.

6.2 RL Model

The PPO model's long-term planning capabilities could be improved by substituting its two MLPs with Recurrent Neural Networks (RNNs) [Goodfellow et al., 2016, p. 373]. With RNNs, the PPO model would have a notion of memory, where its current output is also influenced by its past history of inputs.

Currently, the PPO model is relatively slow to train. For instance, it took 6.5 hours for a model to reach convergence when trained on a single level consisting of seven agents, two cities and a 30×30 grid environment. The current implementation does not support the use of a GPU. OpenAI [2021a] has released a GPU-enabled implementation, named PPO2, which runs three times faster than its CPU-only counterpart. As such, training time could be significantly reduced by redesigning the current implementation to be GPU-enabled.

6.3 Reward Optimization

A new reward function could potentially optimize the final solution. The development would consist of trial and error, and should furthermore focus on avoiding undesirable ways for the agents to achieve rewards. The new reward function could potentially be based on information such as, deadlock avoidance, priority, collaboration, stopping, stopping at a switch, and path length. Additionally, intermediary rewards could also be explored, instead of the sparse rewards supplied by the environment, in order to increase the possible input data and interpret information more often.

The current reward function is solely based on complying with the time schedules. As a consequence, the penalty of an agent being e.g. five time-steps too late, is the same as being deadlocked five time-steps away from the target. In the real world, there is quite a big difference between a train being five minutes late, and a train crashing into another train, five minutes before arrival. Therefore, if the solution should be more applicable to the real world, then the reward function should probably be revised.

The addition of a deadlock parameter and other parameters to the reward function, could potentially add some nuance to the sparse reward function, which could make it more reflective of the solutions' actual performance better. However, as mentioned in Chapter 5, it may result in undesired behavior and would take a long time to implement and a series of continuous testing for intended behavior.

6.4 Performance Evaluation

As described in Section 5.3, the used configuration for the training, validation and testing sets, had some unforeseen consequences, which led to the choice of discarding the test results from the evaluation. The primary reason behind this was the difference between how the training, validation and test set were constructed. In retrospect, some potential improvements have been identified, which can enhance a future evaluation of a TMS for

Flatland. The essential idea behind the modification of the evaluation setup, is to create a set-up where the exact same difficulty levels are used for training, validation and testing. On this matter, the testing environments should naturally use the same environment generation seeds, whereas the training and validation sets should not. This should enable better comparative evaluation of PPO models.

As mentioned in Section 5.3.2, the training set was changed to improve the rate of learning, due to time constraints. This resulted in a dissimilarity, where the training set, for instance, trained on an environment of maximum 16 agents, but the validation was conducted on environments up to 50 agents. The dissimilarity is showcased in Figure 5.14 in Section 5.3.6, where the normalized rewards were approximately at -0.36 at the point of convergence, for the training data in the final solution. In the validation it was only around -0.25, resulting in a difference of around 44%. This indicates that the PPO model is not fully equipped to cope with the increase in difficulty, that is introduced by the validation set. Subsequently, this can result in the performance difference between two solutions being marginalized, as they are not fully trained to cope with the complexity of the validation set. Based on these reflections, if the evaluation had to be reproduced in this project, the validation's levels would be altered to match the levels in the training set, as to reflect the performance better.

In regard to the test setup, a similar conclusion can be drawn, in which the same levels should be used as the training set. As elaborated upon in Section 5.3, the used test set from Flatland, had some unforeseen consequences in regard to the cut-off point after each level at 25% and the few environment instances for each difficulty level. To mitigate this, the cut-off point should be removed in a future evaluation, as it led to solutions seemingly performing significantly better despite it being a result of coincidence. This could be solved by removing the 25% cut-off point. Concerning the low number of environments in each level, it would be preferred to increase the number of environments, in order to reduce the high performance variance.

To summarize, if the above suggestions had been applied to a future evaluation of the solution, the results from the executed evaluation would potentially have a higher coherence between the evaluations. Furthermore, it could result in a more significant performance difference between test results, meaning it can with a higher degree of certainty be concluded that a solution outperforms another.

Conclusion 7

With an increase in transportation capacity within the railway sector in the near future, its traffic is bound to become more complex and difficult to manage. An automated Traffic Management System (TMS) is a prospective solution that can address this increase in complexity and efficiently manage the traffic. Such promise has motivated the investigation of a Reinforcement Learning (RL) based TMS for a simplified 2D grid environment named Flatland. In this regard, the TMS has been viewed as a Multi-Agent Reinforcement Learning (MARL) problem, which has led to the development of a decentralized system, consisting of a network of multiple independently acting TMS agents with a shared policy.

Each agent consists of two components: an RL-model that determines which actions to take, and an observation model that encapsulates the necessary information of the environment. For the RL-model, an actor-critic Proximal Policy Optimization (PPO) architecture has been utilized, as this is considered state-of-the-art. For the observation model, a tree-based observation model has been formed. The finished RL-based TMS was realized through a systematic approach of finding an appropriate training curriculum, determining new features for the observation model, and finally it was optimized with frame-skipping and hyperparameter tuning. In order for the developed solution to be considered applicable to a real world setting, it was ascertained that the RL-based TMS would be measured by its degree of generalizability and scalability to complex railway networks and adaptability to unpredictable events.

On the subject of generalizability, the developed solution has been evaluated on similar randomly generated environment. Despite some inconsistencies in the evaluation results, the general takeaway is that the developed solution is able to generalize from what it has learned during training to similar environments not observed during training. Nevertheless, in order to further solidify this claim, a further inquiry would be necessary.

The developed solution shows some challenges in regard to scalability. However, the evaluation results show that the developed solution is capable of handling the same difficulty levels that it was trained on. Furthermore, the evaluation results indicate that the developed solution to some degree is capable of handling more complex environments than what it was trained on. These findings suggest that the developed solution is scalable within boundaries. Nonetheless, this still remains a theory in need of further examination, where the developed solution needs to be trained with a larger training set containing more complex environments. Additionally, further development is needed in order for it to be considered truly scalable.

In terms of adaptability, additional features have been implemented for the observation model, which sought to improve the handling of malfunctions, deadlocks and varying speed profiles of trains. The evaluation results suggest that the added features has improved upon its adaptability, as the new observation model constituted to a performance increase of 2.4 % and a 20 % increase in completion rate. However, further investigation is still needed in order to better ascertain its adaptability by e.g. evaluating it with a test set with scaling malfunctions and speed profiles.

In final, it can be said that the developed solution, in its current state, is not applicable in a real railway network. Nonetheless, the developed solution manages to show potential in regard to generalization, scalability and adaptability. Furthermore, when compared to other RL-based submissions in the Flatland challenge 3, it performs reasonably well with a 3rd place.

Bibliography

adrianegli, **2020**. adrianegli. *adrian_ egli submission Neurips 2020*, 2020. URL https://gitlab.aicrowd.com/adrian_egli/neurips2020-flatland-starter-kit/-/blob/master/reinforcement_learning/ppo_agent.py. Date: 22/12/2021.

AIcrowd, **12 2021a**. AIcrowd. *Flatland frame skipping and action masking*, 2021a. URL https://flatland.aicrowd.com/research/baselines/action_masking_and_skipping.html. Date: 22/12/2021.

AIcrowd, **12 2021b**. AIcrowd. *Environment Configurations*, 2021b. URL https://flatland.aicrowd.com/challenges/flatland3/envconfig.html. Date: 22/12/2021.

AIcrowd, **12 2020**. AIcrowd. *Flatland evaluation metrics*, 2020. URL https://flatland.aicrowd.com/challenges/neurips2020/eval.html. Date: 22/12/2021.

AIcrowd, **09 2021c**. AIcrowd. *Flatland Environment*, 2021c. URL https://flatland.aicrowd.com/getting-started/env.html. Date: 22/12/2021.

AIcrowd, **12 2021d**. AIcrowd. *Flatland level generation*, 2021d. URL https://flatland.aicrowd.com/environment/level_generation.html. Date: 22/12/2021.

AIcrowd, **09 2021e**. AIcrowd. *Flatland speed profiles*, 2021e. URL https://flatland.aicrowd.com/environment/speed_profiles.html. Date: 22/12/2021.

AIcrowd, **12 2021f**. AIcrowd. *Flatland time limits*, 2021f. URL https://flatland.aicrowd.com/challenges/flatland3/eval.html. Date: 22/12/2021.

AIcrowd, **09 2021g**. AIcrowd. *Flatland timetables*, 2021g. URL https://flatland.aicrowd.com/environment/timetables.html. Date: 22/12/2021.

AIcrowd, **11 2021h**. AIcrowd. *Flatland Environment*, 2021h. URL https://flatland.aicrowd.com/environment/observations.html. Date: 22/12/2021.

AICrowd et al., **2021a**. AICrowd, SBB CFF FFS, SNCF and Deutsche Bahn. *Flatland Challenge*, 2021a. URL https://www.aicrowd.com/challenges/flatland-challenge#background. Date: 22/12/2021.

AICrowd et al., **2021b**. AICrowd, SBB CFF FFS, SNCF and Deutsche Bahn. *Flatland 3*, 2021b. URL https://www.aicrowd.com/challenges/flatland-3. Date: 22/12/2021.

Ang, **2021**. Clifford S. Ang. *Analyzing Financial Data and Implementing Financial Models Using R. Second edition.* Springer, 2021. ISBN 978-3-030-64155-9.

Bishop, **2006**. Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006. ISBN 0-387-31073-8.

Bonvang et al., 2020. Gustav Bonvang, Jonathan Sørensen, Kim Nguyen, Mathias Thorsager, Nicholas Hansen and Sune Krøyer. *Real Time Computer Vision Control System for Autonomous Vehicles*, 2020. Semester project, Aalborg University.

EC, 2019. EC. *Transport in the European Union: Current Trends and Issues*, 2019. URL `https://www.amt-autoridade.pt/media/1934/2019-transport-in-the-eu-current-trends-and-issues.pdf`. Date: 22/12/2021.

Engstrom et al., 2020. Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph and Aleksander Madry. *Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO*, 2020. Date: 22/12/2021.

Fagerlund et al., 2021. Frederik Fagerlund, Gustav Bonvang, Kim Nguyen, Mathias Thorsager and Sune Krøyer. *Optimising Battery Charging Patterns in Satellites Using Recurrent Neural Networks*, 2021. Bachelor Thesis, Aalborg University.

Gharibi et al., 2020. Mirmojtaba Gharibi, Steven L. Waslander and Raouf Boutaba. *Vehicle Scheduling Problem*, 2020. Date: 22/12/2021.

Goodfellow et al., 2016. Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Hare, 10 2019. Joshua Hare. *Dealing with Sparse Rewards in Reinforcement Learning*, 2019. URL `https://www.researchgate.net/publication/336714623_Dealing_with_Sparse_Rewards_in_Reinforcement_Learning`. Date: 22/12/2021.

Henderson et al., 2018. Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup and David Meger. *Deep Reinforcement Learning that Matters*. 2018. URL `https://www.researchgate.net/publication/319953023_Deep_Reinforcement_Learning_that_Matters`. Date: 22/12/2021.

Kingma and Ba, 12 2014. Diederik Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. International Conference on Learning Representations, 2014. URL `https://www.researchgate.net/publication/269935079_Adam_A_Method_for_Stochastic_Optimization`. Date: 22/12/2021.

Kool et al., 2019. Wouter Kool, Herke van Hoof and Max Welling. *Attention, Learn to Solve Routing Problems!*, 2019. Date: 22/12/2021.

Laurent et al., 08 2021. Florian Laurent, Manuel Schneider, Christian Scheller, Jeremy Watson, Jiaoyang Li, Zhe Chen, Yi Zheng, Shao-Hung Chan, Konstantin Makhnev, Oleg Svidchenko, Vladimir Egorov, Dmitry Ivanov, Aleksei Shpilman, Evgenija Spirovska, Oliver Tanevski, Aleksandar Nikov, Ramon Grunder, David Galevski, Jakov Mitrovski and Sharada Mohanty. *Flatland Competition 2020: MAPF and MARL for Efficient Train Coordination on a Grid World*. pages 275–301, 2021. URL `https://www.researchgate.net/publication/354066302_Flatland_Competition_2020_MAPF_and_MARL_for_Efficient_Train_Coordination_on_a_Grid_World`. Date: 22/12/2021.

Li et al., 10 2007. Jing-Quan Li, Pitu Mirchandani and Denis Borenstein. *The vehicle rescheduling problem: Model and algorithms*. Networks, 50, 211–229, 2007. doi: 10.1002/net.20199. URL https://www.researchgate.net/publication/220209620_The_vehicle_rescheduling_problem_Model_and_algorithms. Date: 22/12/2021.

Ma et al., 2011. Hang Ma, T. k. Satish Kumar and Sven Koenig. *Multi-Agent Path Finding with Delay Probabilities*, 2011. URL https://www.researchgate.net/publication/311402397_Multi-Agent_Path_Finding_with_Delay_Probabilities. Date: 12/12/2021.

Mike phillips, 2011. Maxim Likhachev Mike phillips. *SIPP: Safe interval path planning for dynamic environments*, 2011. URL https://ieeexplore.ieee.org/document/5980306. Date: 12/12/2021.

mitchellgoffpc, 2020. mitchellgoffpc. *mitchellgoffpc submission Neurips 2020*, 2020. URL https://github.com/mitchellgoffpc/flatland-training. Date: 22/12/2021.

Mohanty et al., 12 2020. Sharada Mohanty, Erik Nygren, Florian Laurent, Manuel Schneider, Christian Scheller, Nilabha Bhattacharya, Jeremy Watson, Adrian Egli, Christian Eichenberger, Christian Baumberger, Gereon Vienken, Irene Sturm, Guillaume Sartoretti and Giacomo Spigler. *Flatland-RL : Multi-Agent Reinforcement Learning on Trains*. 2020. URL https://www.researchgate.net/publication/346933381_Flatland-RL_Multi-Agent_Reinforcement_Learning_on_Trains. Date: 22/12/2021.

Narvekar et al., 2020. Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor and Peter Stone. *Curriculum Learning for Reinforcement Learning Domains: A Framework and Survey*. J. Mach. Learn. Res., 21, 181:1–181:50, 2020. URL https://www.researchgate.net/publication/339873123_Curriculum_Learning_for_Reinforcement_Learning_Domains_A_Framework_and_Survey. Date: 22/12/2021.

Nazari et al., 02 2018. Mohammadreza Nazari, Afshin Oroojlooy jadid, Lawrence Snyder and Martin Takáč. *Deep Reinforcement Learning for Solving the Vehicle Routing Problem*. 2018. URL https://www.researchgate.net/publication/323141783_Deep_Reinforcement_Learning_for_Solving_the_Vehicle_Routing_Problem. Date: 22/12/2021.

OpenAI, 2021a. OpenAI. *Proximal Policy Optimization*, 2021a. URL https://openai.com/blog/openai-baselines-ppo/. Date: 22/12/2021.

OpenAI, 2021b. OpenAI. *Proximal Policy Optimization*, 2021b. URL https://spinningup.openai.com/en/latest/algorithms/ppo.html. Date: 22/12/2021.

Pishro-Nik, 2014. Hossein Pishro-Nik. *Introduction to probability, statistics, and random processes*. Kappa Research LLC, 2014. https://www.probabilitycourse.com/.

Poole and Mackworth, 2010. David L. Poole and Alan K. Mackworth. *Artificial Intelligence, Foundations of Computational Agents*. Cambridge, 2010. ISBN 978-0-511-72946-1.

Russell and Norvig, **2010**. Stuart Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Pearson, 2010. ISBN 978-9332543515.

Schulman et al., **06 2015**. John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan and Pieter Abbeel. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2015. URL `https://www.researchgate.net/publication/277958927_High-Dimensional_Continuous_Control_Using_Generalized_Advantage_Estimation`. Date: 22/12/2021.

Schulman et al., **07 2017**. John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov. *Proximal Policy Optimization Algorithms*. 2017. URL `https://www.researchgate.net/publication/318584439_Proximal_Policy_Optimization_Algorithms`. Date: 22/12/2021.

Silver et al., **01 2016**. David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel and Demis Hassabis. *Mastering the game of Go with deep neural networks and tree search*. Nature, 529, 484–489, 2016. doi: 10.1038/nature16961. Date: 22/12/2021.

Sutton and Barto, **2018**. Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

Vinyals et al., **2019**. Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, L. Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Caglar Gulcehre, Ziyun Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps and David Silver. *Grandmaster level in StarCraft II using multi-agent reinforcement learning*. Nature, pages 1–5, 2019. URL `https://www.semanticscholar.org/paper/Grandmaster-level-in-StarCraft-II-using-multi-agent-Vinyals-Babuschkin/361c00b22e29d0816ca896513d2c165e26399821`. Date: 22/12/2021.

Wandb.ai, **2021**. Wandb.ai. *Wandb.ai sweeps*, 2021. URL `https://wandb.ai/site/sweeps`. Date: 22/12/2021.

Yu et al., **03 2021**. Chao Yu, Akash Velu, Eugene Vinitsky, Yu Wang, Alexandre Bayen and Yi Wu. *The Surprising Effectiveness of MAPPO in Cooperative, Multi-Agent Games*. 2021. URL `https://www.researchgate.net/publication/349727671_The_Surprising_Effectiveness_of_MAPPO_in_Cooperative_Multi-Agent_Games`. Date: 22/12/2021.

Yu and Zhu, **2020**. Tong Yu and Hong Zhu. *Hyper-Parameter Optimization: A Review of Algorithms and Applications*. abs/2003.05689, 2020. URL

https://www.researchgate.net/publication/339898538_Hyper-Parameter_Optimization_A_Review_of_Algorithms_and_Applications. Date: 22/12/2021.

Čáp et al., 2011. Michal Čáp, Peter Novák, Alexander Kleiner and Martin Selecký. *Prioritized Planning Algorithms for Trajectory Coordination of Multiple Mobile Robots*, 2011. URL https://ieeexplore-ieee-org.zorac.aub.aau.dk/document/7138650. Date: 12/12/2021.

Graphs from the evaluation results

Below is all the graphs that were produced by our evaluation.

A.1 Curriculum

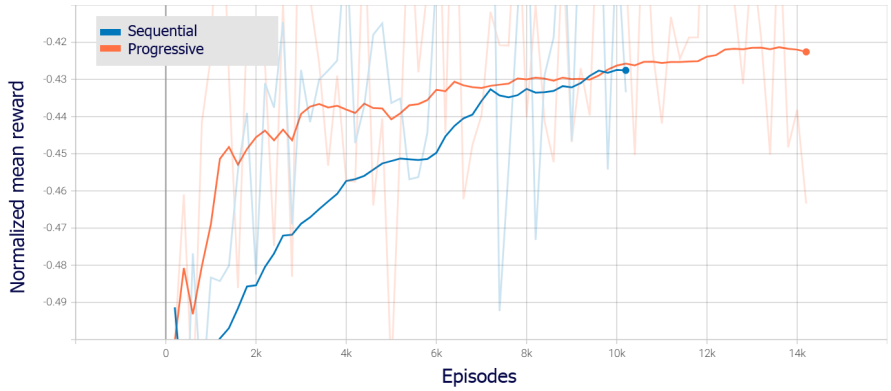


Figure A.1: Validation - normalized mean reward for progressive vs. sequential

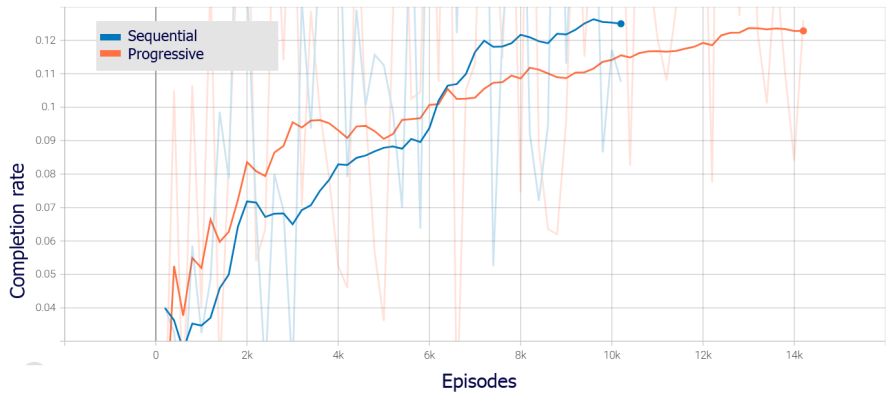


Figure A.2: Validation - completion rate for progressive vs. sequential

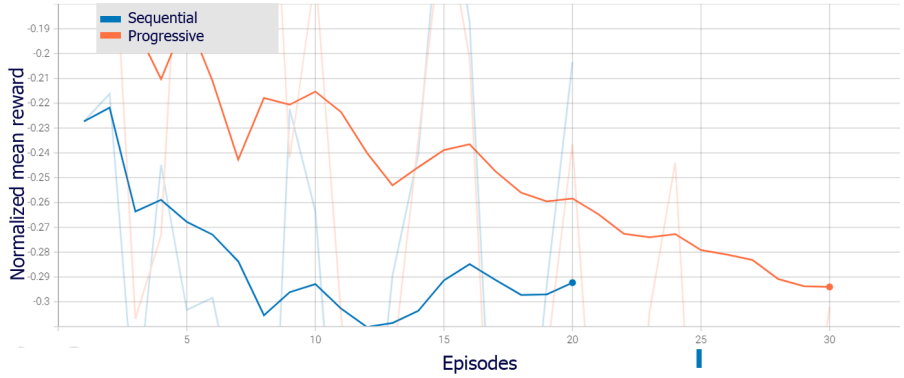


Figure A.3: Test - normalized mean reward for progressive vs. sequential

A.2 Observation

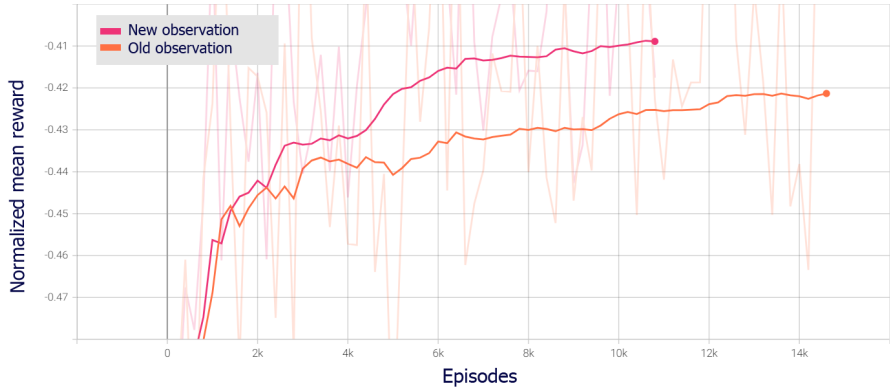


Figure A.4: Validation - normalized mean reward for new observation vs. old observation

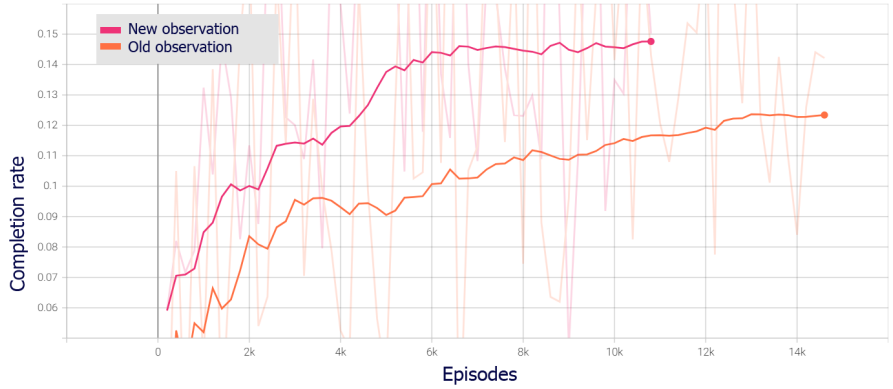


Figure A.5: Validation - completion rate for new observation vs. old observation

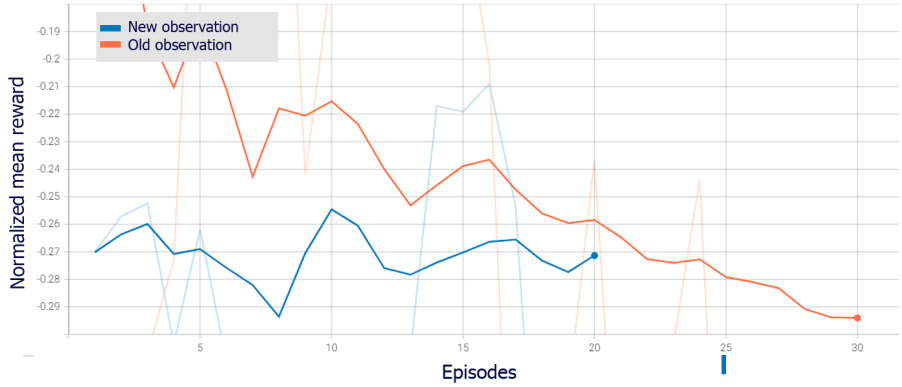


Figure A.6: Test - normalized mean reward for new observation vs. old observation

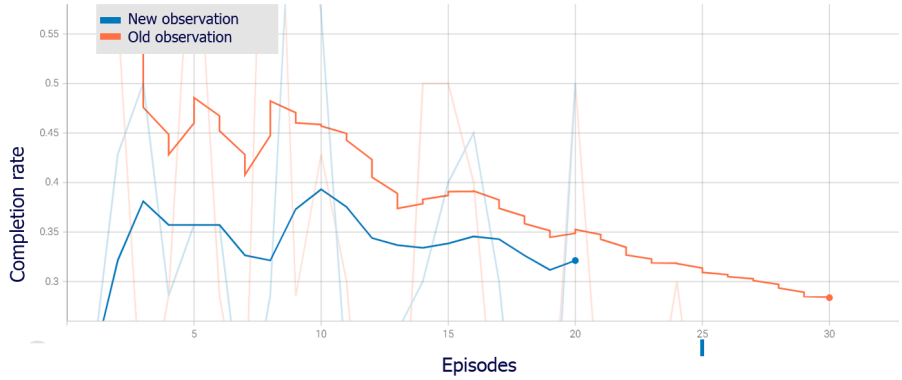


Figure A.7: Test - completion rate for new observation vs. old observation

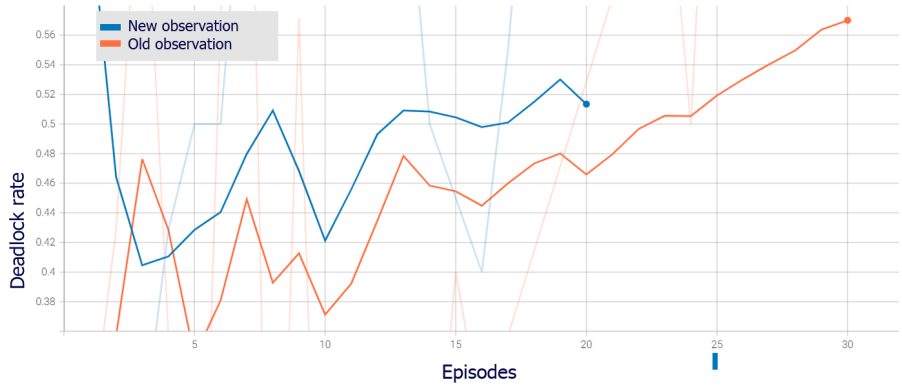


Figure A.8: Test - deadlock rate for new observation vs. old observation

A.3 Frame Skipping and Action Masking

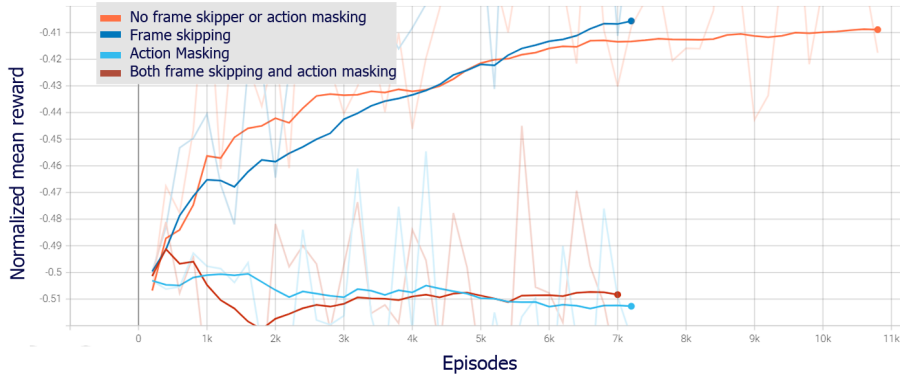


Figure A.9: Validation - normalized mean reward for frame skipping and action masking

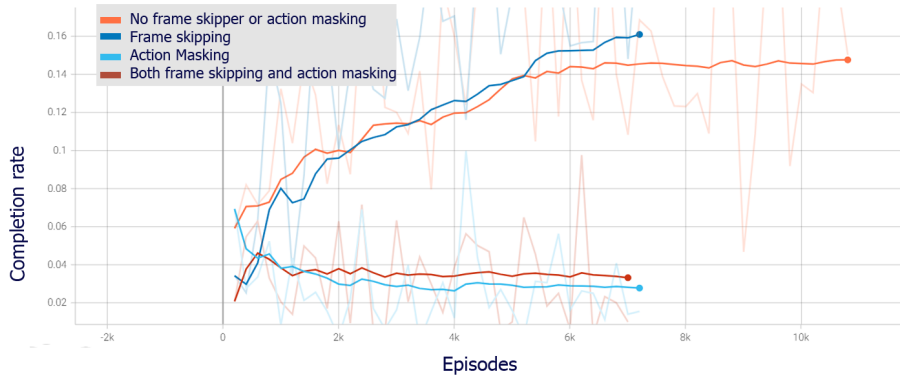


Figure A.10: Validation -completion rate for frame skipping and action masking

Figure A.11: Test - normalized mean reward for frame skipping and action masking

Figure A.12: Test - completion rate for frame skipping and action masking

Figure A.13: Test - deadlock rate for frame skipping and action masking

A.4 Hyperparameter Tuning

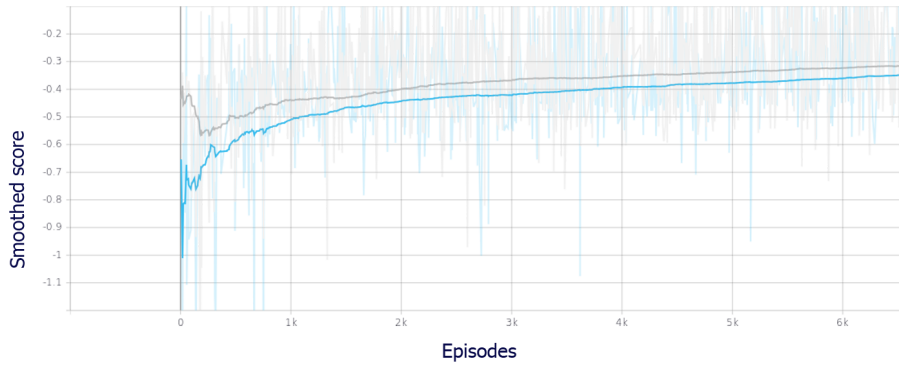


Figure A.14: Training - smoothed score for 2 progressive runs

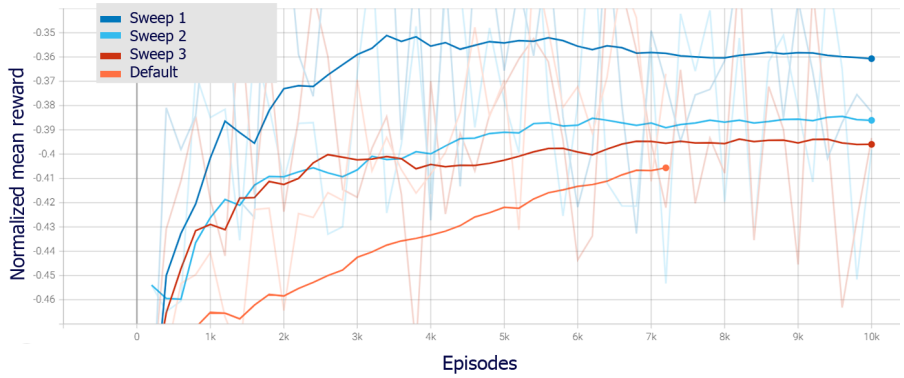


Figure A.15: Validation - normalized mean reward for 3 hyperparameter runs and 1 baseline run

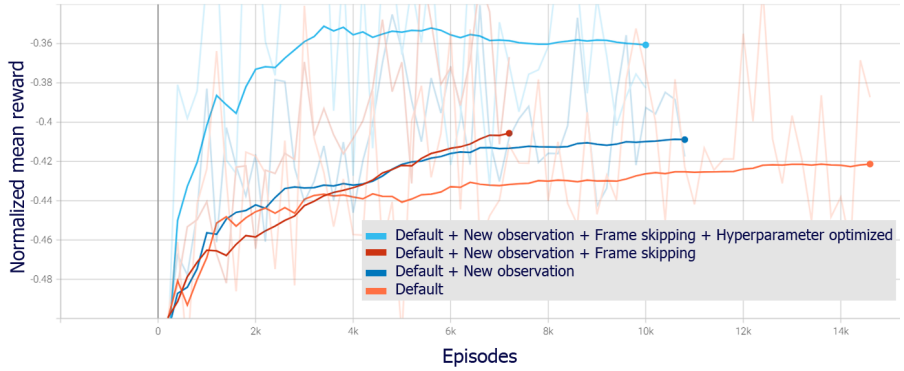


Figure A.16: Validation - normalized mean reward for all steps in the evaluation chapter

A.5 Additional Graphs

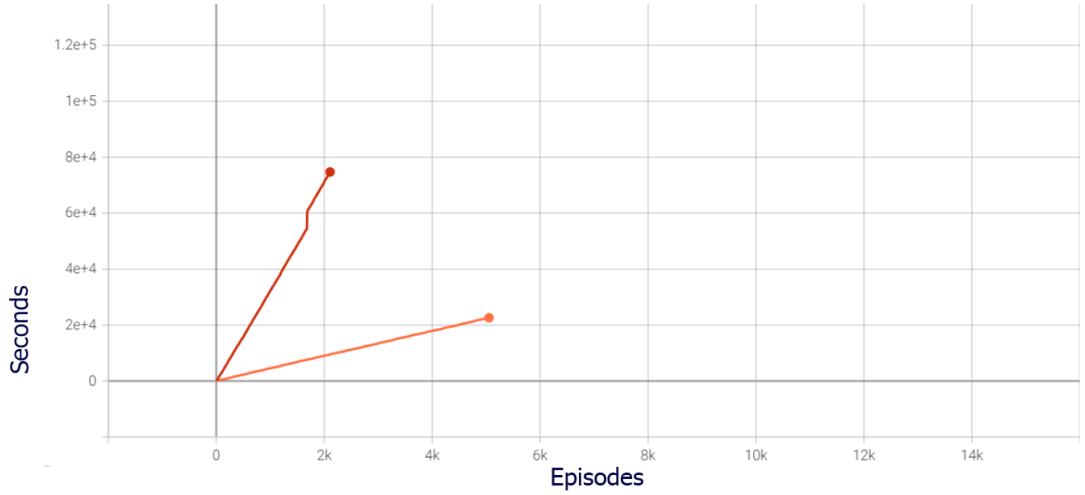


Figure A.17: Training time for sequential (red line) and incremental (orange line) curriculums

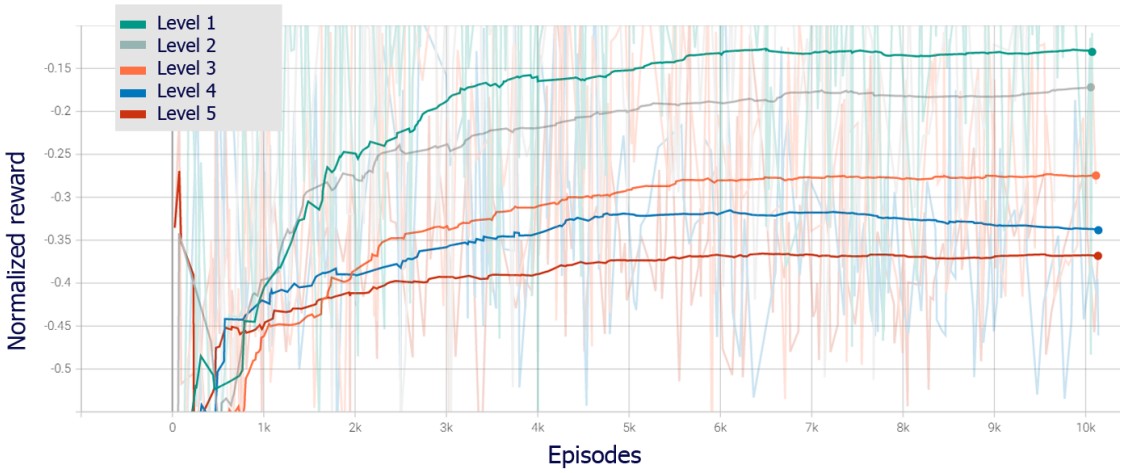


Figure A.18: On the graph is the normalized reward for the 5 difficulty levels displayed, from the final solution's training data

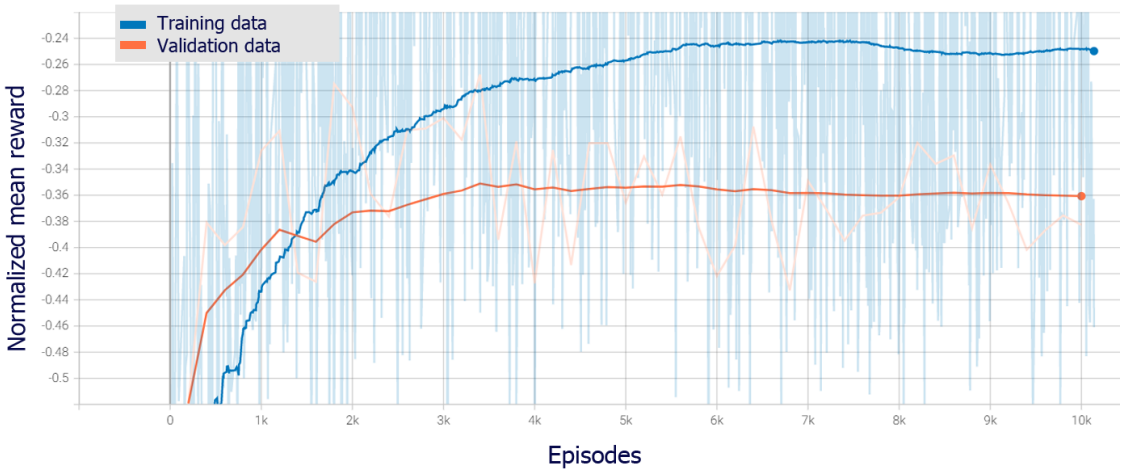


Figure A.19: Normalized mean reward for training vs. validation

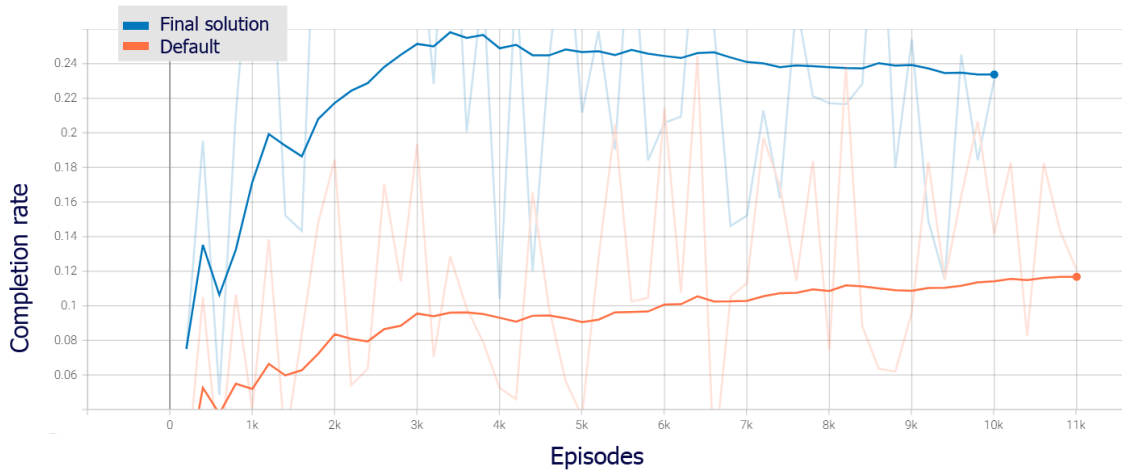


Figure A.20: Completion rate for the default solution vs. the final solution

Various B

Below are various appendices showcased, that helps improve the reader's understanding of central subjects.

B.1 Local Test Cancellation

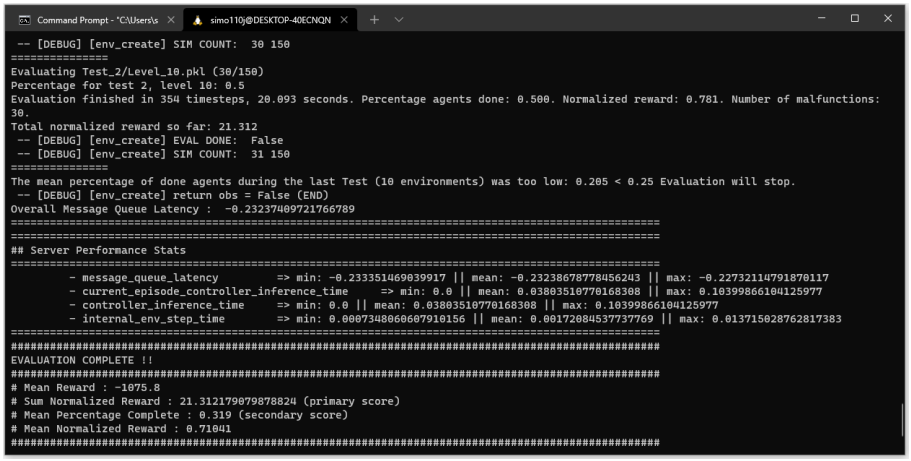


Figure B.1: Cancellation - cancellation in the local testing setup, due to the mean percentage of done agents during the last 10 environments, was below 25%

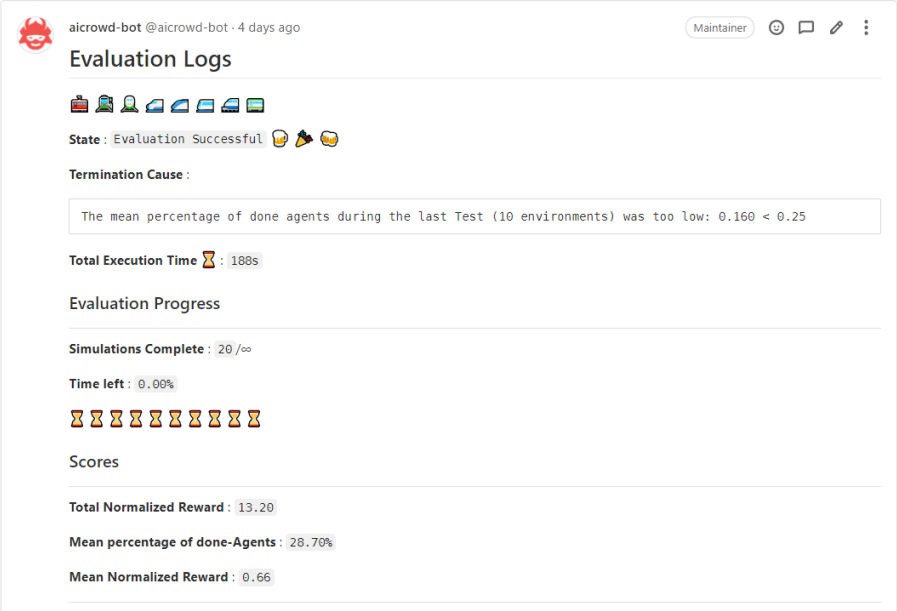


Figure B.2: Cancellation - cancellation on online submission, due to the mean percentage of done agents during the last 10 environments, was below 25%

B.2 Leaderboard: Flatland 3 Challenge

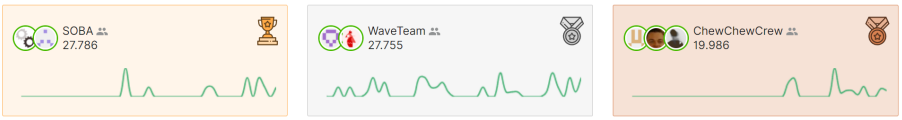


Figure B.3: Leaderboard for the RL track, showcasing the project groups third place in the Flatland challenge. ChewChewCrew is the name of the project group.