Multi-Agent Proximal Policy Optimization on Flatland 3

Edoardo Merli

January 31, 2025

Abstract

Through this project, we aim to test the applicability of Multi-Agent methods to the efficient automated scheduling of trains. To this end, we chose the Flatland 3 challenge, a multi-agent reinforcement learning environment that simulates a railway network, in order to have a setting realistically similar to the real-world problem. We implemented a Multi-Agent Proximal Policy Optimization (MAPPO) algorithm, together with two observations tailored to the problem and two network architectures. We evaluated our trained models and found that our best model would have reached the 1st place on the RL track of the Flatland 3 challenge. Our code is available at https://github.com/edomerli/Flatland-3-MARL.

1 Introduction

The Flatland 3 challenge [2] is a multi-agent environment that simulates a railway network, where agents (trains) need to reach their respective goals while avoiding collisions with each other. The goal of the challenge is to develop efficient routing or scheduling policies that minimize delays. It was first introduced in 2019 as an AICrowd challenge proposed by SBB, the Swiss Federal Railways. Later, it was proposed as a competition in NeurIPS 2020 and AMLD 2021.

Reinforcement Learning (RL) is a type of machine learning in which agents learn to make decisions by interacting with an environment and receiving feedback in the form of rewards or penalties. One of RL's main advantages is its ability to learn complex behaviors by maximizing the reward signal the agent(s) receive, making it a suitable approach for the Flatland 3 challenge. We therefore resorted to using multi-agent reinforcement learning to tackle this challenge.

In section 2 we describe the Flatland 3 challenge. In section 3 we provide our approach, namely implementing Multi-Agent Proximal Policy Optimization (MAPPO), designing custom observations for our agents and a custom reward function, together with two different network architectures. In section 4 we outline the experiments we carried out and in section 5 we report our results. Finally we discuss them and give our final conclusions in sections 6 and 7.

2 The Flatland 3 challenge

2.1 Environment

The Flatland 3 environment is a 2D rectangular grid of arbitrary width and height, where the most primitive unit is a cell. Cells can be rail or grass. Each rail cell can hold a single agent

(train). Each agent moves on rails, with the objective of moving from a starting position to a target position, represented by two stations. Different agents can have different start and target stations, and the goal is to efficiently route each agent to its target station as quickly as possible, while avoiding head-on collisions with other agents. Figure 1 below shows an example of a Flatland 3 environment.

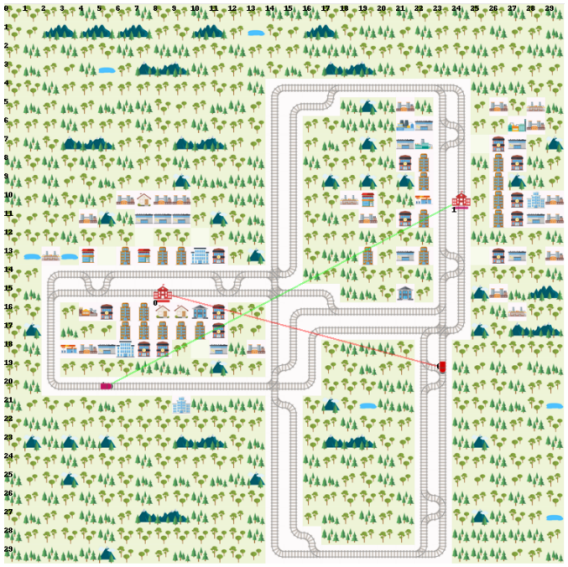

Figure 1: An example of a Flatland 3 environment map. The two trains have to reach their respective target stations, represented by the red buildings they are connected to (the lines are represented only for visualization purposes).

An agent in a cell has a discrete orientation that represents its cardinal direction (North, East, South, West), and can only travel in the direction it's currently facing. The time is divided into discrete timesteps (from 0 to T_{max}, defined depending on the size of the environment), and agents take actions at each timestep to move along the rails. At each step, the agent can move to a subset of adjacent cells depending on the type of rail cell it is currently on and its direction.

Figure 2 below shows the different types of cells in the environment. To ease comprehension: only the cells containing the word "switch" enable the agent to take more than one possible route. The others ("diamond crossing" included) have only one possible route.

It's important to note that agents cannot go backward. Two agents that end up one in front of the other with opposite directions will end up in a deadlock and stall until the end of the episode.

2.2 Action space

The discrete set of actions that an agent can take consists of the following actions:

- **Do nothing**, i.e. if the agent is already moving, it continues moving; if it is stopped, it stays stopped.

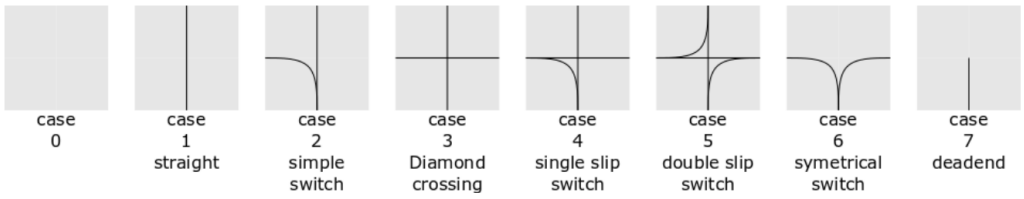

Figure 2: Different types of cells in the Flatland 3 environment. The cells with the word "switch" enable the agent to take more than one possible route, while the others have only one possible route. Curves are a special type of case 1 transition maps where the transition includes a 90-degree rotation.

- **Turn left**, only valid on cells with a switch that enables the agent to change direction to the left.
- **Move forward**, following the direction the agent is facing (turns automatically if the cell is a curve).
- **Turn right**, only valid on cells with a switch that enables the agent to change direction to the right.
- **Stop**, useful to avoid collisions or to wait for other agents to pass.

If an agent takes an action that is not allowed, it will be substituted with the "do nothing" action.

2.3 Agent attributes

Each agent has the following attributes:

- **Earliest departure time**: the earliest timestep the agent can depart from its starting station.
- **Latest arrival time**: the latest timestep at which the agent must arrive at its target station. If the agent does not reach its target station by this time, it will be penalized.
- **Speed**: the number of cells the agent can move in a single timestep. The speed is constant for each agent and is one of the following values: 0.25, 0.33, 0.5, 1.0 (i.e. the agent can move once every 4, 3, 2, or 1 timesteps, respectively). This addition to the environment is meant to simulate the different speeds of trains in a real-world railway network.
- **Malfunctions**: the agent can experience malfunctions that cause it to stop moving for a certain number of timesteps. The duration of the malfunction is randomly sampled from a uniform distribution between a minimum and maximum value, and occurr with a certain probability at each timestep.

2.4 Observations

The environment is fully observable, meaning that agents can see the entire grid, including the positions of other agents, the rails and each agent's target station. At each timestep, each one of

the N agent receives an observation that can include information about its state and speed, the surrounding grid cells and the positions and attributes of other agents. The developers have full control over the observations that agents receive, and can customize them as they see fit.

2.4.1 Provided observations

Three observations, also represented in Figure 3 below, are provided as starting points:

- **Global observation**: the entire grid is observed, including the positions of all agents, their attributes, the rails, and the target stations.
- **Local observation**: similar to the global observation, but only a local portion of the grid around the agent is observed
- **Tree observation**: each agent receives a tree observation that includes information about the possible cells it could encounter in the near future along his route, organized in a tree-like structure.

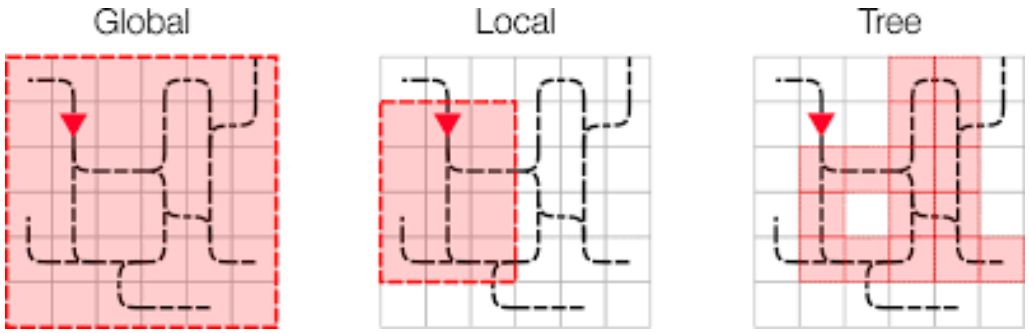

Figure 3: The three types of observations provided to the agents in the Flatland 3 environment.

2.5 Reward function

In Flatland 3, rewards are provided only at the end of the episode, making it a sparse reward environment. The episodes end when all agents have reached their target stations or when the maximum number of timesteps is reached. The reward function is designed to encourage agents to reach their goals as quickly as possible while minimizing delays. Agents receive a reward equal to 0 if they reach their target station before the latest arrival time, receive a penalty proportional to the delay if they arrive after the latest arrival time, and receive an additional penalty (proportional to their distance) on top of the previous one if they do not reach their target station by the end of the episode.

More formally, the reward function for an agent i that has latest arrival time A_i and arrives at timestep T_i is defined as:

$$R_i = \begin{cases} 0, & \text{if } T_i \leq A_i \\ A_i - T_i, & \text{if } A_i < T_i \leq T_{\max} \\ A_i - T_{\max} - d_i^{(T_{\max})}, & \text{otherwise} \end{cases}$$

where $d_i^{(T_{\max})}$ is the time the agent would take, starting from its current position, to reach its target station at timestep $T_{\max}$, if traveling on the shortest path.

The test metric used to evaluate the performance of the routing policies is the following:

$$\bar{R} = 1 + \frac{\sum_{i=1}^{N} R_i}{N \cdot T_{\max}} \in [0, 1]$$

3 Our approach

3.1 Multi-Agent Proximal Policy Optimization (MAPPO)

We modeled this problem as a Multi-Agent Reinforcement Learning problem, where each train is an agent. We implemented a Multi-Agent version of the RL Proximal Policy Optimization (PPO) algorithm [6], refered to as MAPPO [8].

3.1.1 PPO

In PPO, we learn a policy $\pi_\theta(a|s)$, i.e. a distribution of actions to take by the agent given the state it is in, and a value function $V_\phi(s)$, estimating the discounted future cumulative reward received by following π_θ from state s. Both are represented as two neural networks.

TRPO [4] introduced the loss function

$$L^{TRPO}(\theta) = -\hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right]$$

The ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ makes it so that actions that are more probable to be taken under the updated policy, compared to before the optimization, weight more in the loss, and also the opposite (relatively less probable actions are down-weighted). $\hat{A}_t$ is the advantage function, i.e. it models the future cumulative reward and can do so in many different way as a variety of advantage functions have been proposed over the years. We decided to use Generalized Advantage Estimation, or GAE [5], as it's the most popular choice in the literature.

The main innovation introduced by the PPO algorithm is the *Clipped Surrogate Objective Function*, i.e.

$$L^{CLIP}(\theta) = -\hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

To avoid updates with too drastic changes of actions distributions from the frozen policy, the ratio is clipped (usually $\epsilon = 0.2$, as in our case). This helps to have more stable updates, and for this reason, makes it possible to train the network for multiple epochs on the same set of collected transitions.

The resulting algorithm is the following:

Algorithm 1 PPO

Input: Initialized π_θ, $\pi_{\theta_{\text{old}}}$, V_ϕ
  for $iter = 1, \ldots, N$ **do**
    Collect T transitions by following π_θ
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$ using V_ϕ
    for $epoch = 1, \ldots, M$ **do**
      Optimize $L^{CLIP}(\theta)$ wrt θ, using π_θ and frozen $\pi_{\theta_{\text{old}}}$
      Optimize $L^{MSE}(\phi)$ wrt ϕ with $\hat{A}_1, \ldots, \hat{A}_T$ as targets
      $\pi_{\theta_{\text{old}}} \leftarrow \pi_\theta$
    end for
  end for

3.1.2 MAPPO

In MAPPO, the policy π_θ takes as input the individual local observation of an agent and returns an action distribution (although we also experimented with messaging mechanisms), while the value function $V_\phi(s)$ takes global information as the union of the local informations of all agents and its output is used as in PPO as a baseline for variance reduction.

3.2 Binary observations

At each timestep t, each agent i receives an observation $o_i^{(t)}$ that includes information about its state and speed, the surrounding grid cells, and the positions and attributes of other agents along its track. Between the three observations provided by Flatland (global, local, and tree), we chose to use the tree observation as a starting point, as it was the recommended one and the one that seemed to best capture the relevant information for the agents to make decisions. Nonetheless, early experiments showed that the tree observation provided by Flatland was too time-consuming to produce by the environment and to process by the neural network (due to its high dimensionality) for the computing resources we had at hand, so we decided to design a custom observation that would be more efficient to compute and process. We called this a **binary observation**, which is similar in structure to the tree observation (albeit smaller in size) but containing only binary information.

It consists of two components, one for the agent's attributes and one for the grid cells in the agent's possible paths up to a depth of two switches. We designed two versions of this observation: the first one has a smaller dimensionality (40) because it stores information regarding the whole subtree by aggregating over the second depth, at the cost of not knowing which depth two rail branch contains which other agent or target station; the second one has a bigger dimensionality (174) because it stores depth two branch information explicitly without aggregation. Tables 1 and 2 give a detailed description of the two versions of the binary observation, while Figure 4 shows two examples of environment states with the respective visited cells in the observation for each agent highlighted. Note how the exploration of the cells in the agent's possible paths stops when an opposite agent is encountered, as encoding the ordering of encounters of more than one agent (or station) while remaining in the binary domain would have made the observation too complex.

We implemented a deadlock detection algorithm to obtain deadlocks information.

	Description	Dim.
Agent features	Train state (one-hot encoded)	7
	Agent is at switch	1
	Agent is near any switch	1
	Agent is one step before a decision switch	1
	Agent is one step before any switch	1
	Agent's speed (one-hot encoded)	4
	Agent is deadlocked	1
	for each one of the 4 directions:	
Subtree features	there is a path towards the target	1
	that path is the shortest path to the target	1
	there is an opposite-direction or deadlocked agent	1
	there is a same-directiona and not deadlocked agent	1
	the agent's target is on this path	1
	there is another agent's target on this path	1

Table 1: Binary observation v1. A decision switch is a switch at which the current agent can take a decision. The subtree features refer to the subtree of depth two obtained by branching at each switch in which there is more than possibile decision.

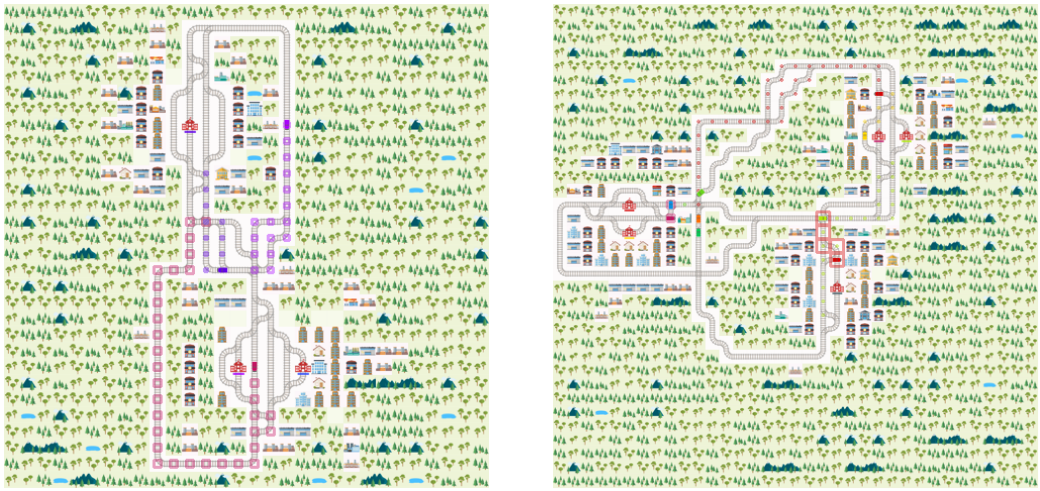

Figure 4: Two environment states with the respective visited cells in the observation for each agent highlighted, with so called "demo" setup (5 agents, left) and "mini" setup (20 agents, right). Note how as agents on the map raise in numbers, the traffic increases and deadlocks start to occur.

	Description	Dim.
Agent features	Train state (one-hot encoded)	7
	Agent is at switch	1
	Agent is near any switch	1
	Agent is one step before a decision switch	1
	Agent is one step before any switch	1
	Agent's speed (one-hot encoded)	4
	Agent is deadlocked	1
Subtree features	for each one of the 4 directions:	
	for each one of the 4 sub-branches	
Subbranch features	there is a transition	1
	there is a path towards the target	1
	that path is the shortest path to the target	1
	there is an opposite-direction agent	1
	there is a same-direction agent	1
	the same-direction agent is at least as fast as this one	1
	the other agent is deadlocked	1
	the other agent is in a "ready to depart state"	1
	the other agent is malfunctioning	1
	the agent's target is on this path	1

Table 2: Binary observation v2. A decision switch is a switch at which the current agent can take a decision. The subtree features refer to the subtree of depth two obtained by branching at each switch in which there is more than possibile decision.

3.3 Neural network architectures

For the policy and value networks, we implemented two different neural network architectures: a Multi-layer Perceptron, more simple and efficient, but not capable of any message passing primitive, and a Transformer architecture, named RailTransformer, more complex and computationally expensive but potentially capable of learning message passing mechanisms through the attention mechanism.

The policy and value networks possess the same architecture (apart from the number of output neurons, 5 as the number of actions for the policy and 1 for the value network) but in two distinct networks with no parameter sharing, as [1] has shown that it resulted in substantial improvement in performance over the parameter sharing counterpart.

As highlighted by [1], the initial policy has a high impact on training performance. They suggest initializing the last layer of the policy network with small weights, such that the initial action is almost independent of the observation. We do so by setting them to be orthogonal and with a value of 0.01, on top of setting bias to 0. We did the same for the value network but with the initial value set to 1.0.

We also added residual connections after every fully-connected hidden layer, as [3] has shown it to

benefit training.

3.3.1 Multi-layer Perceptron (MLP)

A Multi-layer Perceptron with two fully-connected hidden layers of dimension 128 and tanh activation functions.

The MLP is applied indipendently on each agent's observation, enabling the architecture to work with environments of arbitrary number of agents.

3.3.2 RailTransformer

All the agents' observations are processed together in this architecture, each one being considered as a *token* in the standard NLP interpretation of the Transformer [7]. Every token/observation is first processed by a 3-layers MLP (128 hidden dimension, ReLU activations and residual connections), which "embedds" it to then go through 3 Transformer encoder blocks (4 heads each and feedforward expansion ratio set to 4).

In the policy network case, the resulting embeddings are then processed again by a 2-layers MLP identical to the first one and finally projected to the action space and normalized to sum to 1, obtaining a probability distribution over the set of actions for each agent.

In value network case, to obtain a single value, we prepended a special "value token" to the initial input sequence and then apply the final 2-layer MLP only to that token's embedding after the Transformer phase.

Figure 5 shows a diagram of this architecture for the policy network.

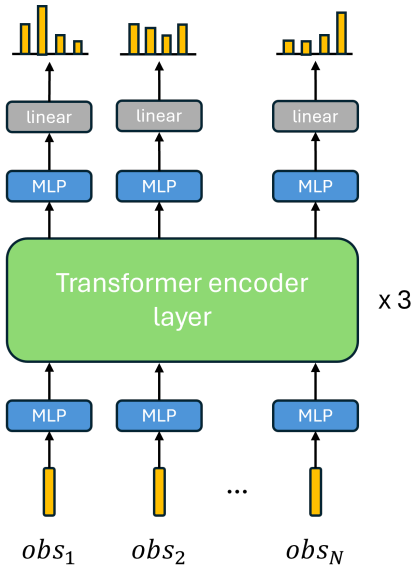

Figure 5: RailTransformer architecture for policy network case.

3.4 Custom reward function

In RL tasks, it's often beneficial to design a dense custom reward function tailored to the desired agent behaviour, which will be optimized as a proxy of the sparse true reward function to ease learning. In our case, we created our custom reward function R_t (at time t) as following:

$$\hat{A}_t = \sum_{i|done_i^{(t)}} 1 - \frac{max(t - l_i, 0)}{T_{max}}$$

$$R_t = +0.1\frac{D_t}{N} + 5\frac{\hat{A}_t}{N} - 2.5\frac{L_t}{N}$$

where D_t is the number of agents departed at timestep t, $done_i^{(t)}$ is true if agent i arrived at its destination exactly at timestep t, l_i is the latest arrival time for agent i, $\hat{A}_t$ is the number of agents arrived to their respective destination at timestep t with a penalty for arriving late and L_t is the number of newly deadlocked agents at timestep t. N is the number of agents in the environment, used for normalization.

The reward is shared by all agents in order to encourage cooperation between them.

4 Experiments

The detailed list of environments configuration hyperparameters used in training and evaluation can be found as csv files in the respective folder[1] of our repo.

4.1 Training setup

We trained our models on Kaggle, using the P100 GPU provided. All the training hyperparameters for our method can be found in Appendix A.

For the environment configurations hyperparameters, we initially considered 5 environments of increased size and number of agents (named "demo", "mini", "small", "medium", "large", respectively with 5, 20, 50, 80 and 100 agents), with hyperparameters identical to those used by similar-sized environments used for evaluation (see subsection 4.2 below). The only main difference is the malfunction interval, fixed to 1000 here.

As we saw our models failing to learn anything meaningful from the "medium" training environment onward (due to the excessive difficulty of handling such a large number of agents in the rail network), we chose to limit ourselves to the first 3 environments for training.

We trained every combination of network architecture (MLP, RailTransformer), observation version (v1, v2) and training recipe (either training from scratch or using curriculum learning by starting training from the previous environment's last checkpoint) for the "demo", "mini" and "small" training environments. The results are displayed in Table 3.

[1] `https://github.com/edomerli/Flatland-3-MARL/tree/main/env_configs`

4.2 Evaluation setup

Initially we wanted to submit our solution to the Flatland 3 evaluator on AIcrowd, but upon doing so, the evaluation failed without providing any logs, making debugging impossibile. Nonetheless, the submission code worked locally, so we resorted to using the flatland-evaluator package from the Flatland 3 library. We obviously used the same test environments configurations, kindly provided by the organizers in the challenge page [2].

As not all environments hyperparameters are given explicitly, we set the remaining ones following the test environments configurations of the AMLD 2021 and NeurIPS 2020 versions of the challenge. More info regarding the test environments hyperparameters are available in Appendix B.

When testing, we used the last policy checkpoint (as the value network is not needed at test time) from the training environment having at least that amount of agents, e.g. for a test environments with 7 agents we used the "mini" training environment checkpoint (20 agents) and not the "demo" one (5 agents). The results are displayed in Table 4.

We provided an updated leaderboard[3] of the submissions to the RL track of phase 2 of the AIcrowd Flatland 3 challenge in 5.

4.3 Additional experiments

Initially, we experimented also with adding Value target normalization to PPO and skipping training on the steps in which no train can take a decision (note that departing from a station is a decision that the agent has to take).

Additionally, we validated our choice of using a custom reward function and its design by comparing a few runs with the custom reward proxy against optimizing directly the environment true reward function.

The plots for these additional experiments can be found in Appendix C. As can be seen, value target normalization and skipping no-choice steps didn't provide any benefits, and in fact worsened performance, so we run our experiments without those. Furthermore, our custom reward function has proven to consistently improve performance in these experiments, even more on larger environments.

[2] `https://flatland.aicrowd.com/challenges/flatland3/envconfig.html`

[3] the original leaderboard can be found at `https://www.aicrowd.com/challenges/flatland-3/leaderboards?challenge_leaderboard_extra_id=1023&challenge_round_id=1083`

5 Results

5.1 Training

model	demo		mini		small	
	true episodic reward	percentage done	true episodic reward	percentage done	true episodic reward	percentage done
MLP + obs v1	0.84	0.67	0.72	0.36	(curr) 0.58	(curr) 0.058
MLP + obs v2	0.89	0.78	0.70	0.34	(curr) 0.59	(curr) 0.081
RailT + obs v1	0.81	0.62	0.61	0.19	0.55	0.043
RailT + obs v2	Train Div.	Train Div.	0.62	0.21	0.54	0.023

Table 3: **mean** true episodic reward and percentage of agents that reached their destination by different model and observation choices. We display metrics for the best training runs between curriculum training and training from scratch. (curr) refers to curriculum performing better than training from scratch. For each training environment, we highlighted the best performing model-obs pair in gold and the second best performing one in silver. Training Div stands for "training diverged", i.e. all metrics collapsed to 0.

5.2 Testing

model	sum normalized reward	mean percentage done
MLP + obs v1	29.22	0.36
MLP + obs v2	28.42	0.34
RailTransformer + obs v1	20.34	0.28
RailTransformer + obs v2	21.99	0.39

Table 4: The sum normalized rewards refers to the reward obtained summed across the test environments. The evaluator stops if the mean percentage of done agents during the last test (i.e. 10 environments) drops below 25%.

5.2.1 AIcrowd challenge comparison

participant	sum normalized reward	mean percentage done
our (MLP + obsv1)	29.22	0.36
WaveTeam	27.868	0.386
SOBA	27.786	0.326
ChewChewChew	19.99	0.306
traintrains	6.71	0.00
g5dna	6.71	0.00

Table 5: Comparison of our test results with those submitted to the **phase 2** of AIcrowd Flatland 3 challenge (already concluded at the current moment) for the **RL track**, resulting in an updated leaderboard. The sum normalized reward is the primary score and the percentage done is the secondary score for the leaderboard. Gold, silver and bronze medals have the respective colors.

6 Discussion

In the training regime, shown in Table 3, the MLPs seem to be outperforming the RailTransformers, either due to the RailTransformer architecture having excessive capacity for the task or having a much more gradual learning curve and therefore reaching worse metric values in the fixed training steps budget.

Interestingly, curriculum learning seems to be beneficial only for MLPs and only when the environments start reaching a certain size.

The MLP models keep outperforming the RailTranformers even in the testing environments, as seen from table 4. Finally, our method beats all the other submissions to the RL track of Flatland 3, as can be seen from 5.

7 Conclusion

In this project we implemented a Multi-Agent Proximal Policy Optimization (MAPPO) algorithm, together with two observations tailored to the problem and two network architectures. The trained models outperformed all the other submissions to the RL track of the Flatland 3 challenge, hypothetically reaching the 1st position.

As the challenge let people submit also non-RL solutions, among which Operations Research ones got scores even 4 times higher than the RL ones, there is still room for improvement on the RL track of this challenge. To address the limitation of our code of training on one environment at a time, parallel trajectory collection could be beneficial, enabling the model to train at a faster rate as the trajectory collection is the bottleneck of the PPO algorithm in our case.

Overall, we are happy to have reached a good albeit unexpected result in such a complex challenge and with limited computational resources, on top of having learned a lot by implementing a state of the art algorithm like PPO.

References

[1] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphael Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. 2020. What matters in on-policy reinforcement learning? a large-scale empirical study.

[2] 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. 2020. Flatland-rl : Multi-agent reinforcement learning on trains.

[3] A. Emin Orhan and Xaq Pitkow. 2018. Skip connections eliminate singularities.

[4] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. 2017. Trust region policy optimization.

[5] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2018. High-dimensional continuous control using generalized advantage estimation.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention is all you need.

[8] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. 2022. The surprising effectiveness of ppo in cooperative, multi-agent games.

A Implementation details and hyperparameters

Hyperparameter	Value
Total steps	4M
# steps per iteration	1024
Epochs per iteration	10
Batch size	128
Learning rate	2.5×10^{-4}
LR schedule	Cosine
Adam ϵ	1×10^{-5}
action size	5
hidden size	128
Skip no choice cells	False
PPO clip ϵ	0.2
PPO γ	0.999
PPO λ	0.95
Separate networks	True
Value targets normalization	False
Entropy loss coefficient	1×10^{-2}
KL divergence limit	0.02 (MLP) / 0.05 (RailTransf)

Table 6: Hyperparameters used for training

B Evaluation environments configuration

Similarly from AMLD 2021 and NeurIPS 2020, we set:

- max_rails_between_cities = 2
- grid_mode = False
- malfunction_duration = [20, 50]
- malfunction_interval = 250 * (env_id+1), i.e. 250, ..., 2500, just slightly different from the 0, ..., 2250 of AMLD 2021 and NeurIPS 2020

Additionally, we chose: max_rail_pairs_in_city = 2, to train on the hardest scenario.

The seeds are sampled randomly between 1 and 1000.

C Additional experiments plots

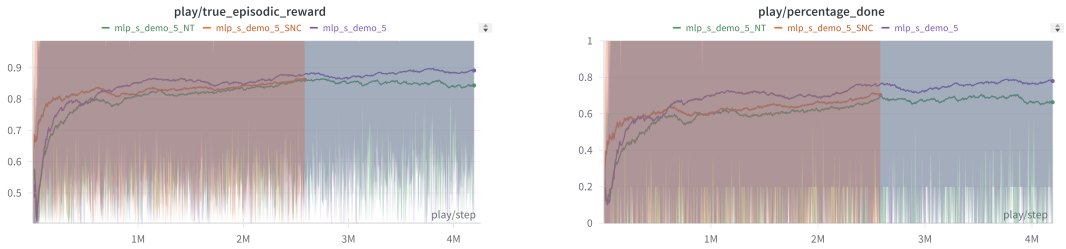

Figure 6: True episodic reward (top) and percentage done (bottom) on the "demo" environment between the base MLP model, Target Normalization added (NT) and skip no-choice cells added (SNC). The base model performs best across both. Best viewed on screen.

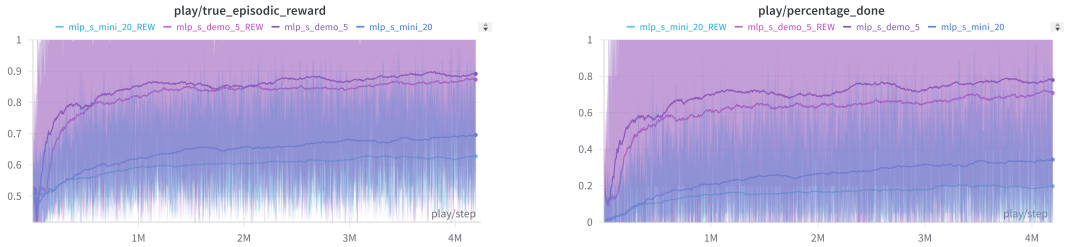

Figure 7: True episodic reward (top) and percentage done (bottom) on the "demo" and "mini" environments between the base MLP model with and without (REW stands for without) the custom reward function. The custom reward function performs best across both. Best viewed on screen.