Alma Mater Studiorum - University of Bologna

Flatland challenge as Deep Learning exam project

Professor
Andrea Asperti

Students
Simone Giordani - 982933
Emmanuele Bollino - 985420
Agostino Aiezzo - 982514

Indice

Introduction

The Flatland challenge is a competition organized by AIcrowd in collaboration with the SBB (Swiss Federal Railways) which manages the densest mixed railway traffic in the world and maintains the biggest railway infrastructure in Switzerland. In fact, there are more than 10 000 trains running each day, being routed over 13 000 switches and controlled through more than 32 000 signals.

The aim of the challenge is to achieve innovation for what concerns the scheduling of trains trajectories in railway environments by solving the 'vehicle rescheduling problem' (VRSP): it arises when a previously assigned trip is disrupted (because of traffic accidents, medical emergencies, breakdown of a vehicle, etc.).

For this reason, the report is organized as follows: first of all, there's an overview of the environment in which each of its components is described. Eventually, all changes about a specific component made in order to get good solutions are reported. Then, in the second part, the proposed solution is presented and justified: the architecture of the model is deepened, as well as a comparison between 2 possible methodologies as regards the exploitation / exploration part of the model itself. In addition all the results are analyzed while, in the third and last part, the sums are drawn.

Flatland environment

In order to conceive a feasible solution for this challenge, it is necessary to fully understand the flatland environment: which are its characteristic elements, how it works, what is possible to modify, etc. To do that, the Python package 'flatland-rl version 2.2.2' has been studied and tested.

Railway environment

The 'RailEnv' class is one of the fundamental bases of the above package. In particular, it returns an object that is a simplification of a railway environment as a 2D grid, in which multiple agents (trains) with different target destinations (stations) must collaborate to maximize the global reward and minimize the travel time for each of the agents, avoiding bottlenecks.

In order to define a 'RailEnv' object, the following values need to be provided:

- **width:** the width of the 2D grid.
- **height:** the height of the 2D grid.
- **number_of_agents:** the number of trains in the 2D grid.

Other additional parameters can be used to extend and better define the railway environment. The most relevant ones are:

- **rail_generator:** it's the component which deals with the layout of the railway, the agents and the targets.
- **schedule_generator:** it's the component which, given the layout of the map, takes care of assigning to each agent a target and a speed value.
- **obs_builder_object:** it's the component which takes care of generating the observations of the environment.
- **malfunction_generator_and_process_data:** it's the component which deals with the generation of the malfunctions for the agents.

Since, in Flatland, time is discrete, in order to advance and see progress in the map, the 'step()' function must be applied on an object of class 'RailEnv'. In particular, given a dictionary containing, for each agent, the action that needs to be performed, it returns many information:

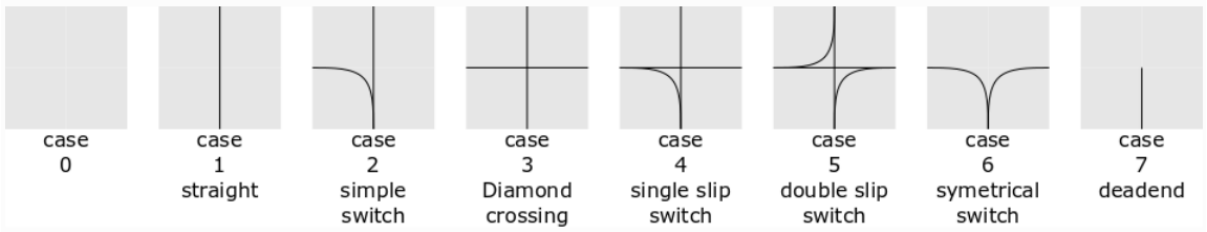
- **obs:** the observations of the environment generated by the observation_builder.

- **rewards:** a dictionary that associates a reward to each agent.
- **dones:** a dictionary that points out with a boolean value which are those agents that have reached their destination. An additional key, '__all__', is set to True when all the agents have concluded their task.
- **info:** a dictionary that contains information about the agents.

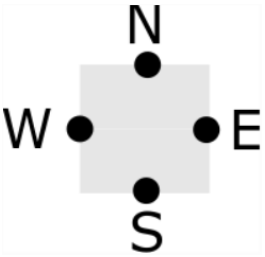
In conclusion, there are other functions that allow you to graphically visualize the environment during a simulation, with all the agents that move along the railway, the targets, and so on. Here is an example:

Map

As we said at the beginning, the map of a railway environment is a bidimensional grid where each position corresponds to a cell that can be occupied by at most one agent or a target station. There are many types of cells:



As we can notice, each cell can have a maximum of 4 inputs/outputs corresponding to the 4 cardinal points:



In addition, these types of cells can be rotated 0, 90, 180, 270 degrees in order to build more complex and realistic railways.

Agents

An agent is an entity whose aim is to reach its own target in the shortest possible time. Each agent is identified by a progressive ID, a speed value, a state and a certain set of actions to choose from.

The movements of an agent on the grid can be carried out only between two adjacent cells, limited by the possible movements which are defined according to the type of the cell and the input cardinal point of the agent.

In fact, each agent is characterized by 5 possible actions:

- **DO_NOTHING (0):** if the agent is stationary, it stays in place; if the agent is moving, it continues to move.
- **MOVE_LEFT (1):** in the presence of a switch, the agent turns left; if the agent is stationary, it starts to move.
- **MOVE_FORWARD (2):** in the presence of a switch, the agent goes straight; if the agent is stationary, it starts to move.
- **MOVE_RIGHT (3):** in the presence of a switch, the agent turns right; if the agent is stationary, it starts to move.
- **STOP_MOVING (4):** if the agent is moving, it stops.

Invalid actions do not have any effect.
Just to simplify a bit the navigation in the search space, the ‘DO_NOTHING’ action has been removed, since its presence does not influence the learning of the model.

For what concerns the state of an agent, it is an attribute whose possible values are:

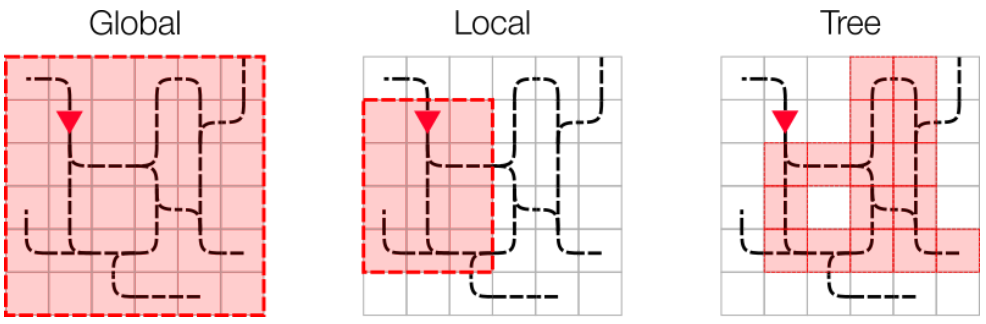
- **READY_TO_DEPART (0):** the agent hasn't left yet, so it is not on the initial rail.
- **ACTIVE (1):** the agent is active and it is on a rail
- **DONE (2):** the agent has arrived at its destination, but it is still on the arrival rail.
- **DONE_REMOVED (3):** the agent has arrived at its destination and it has been removed from the map.

To simulate different speeds, each agent moves at a variable pre-assigned velocity that can be unitary (1 cell at each timestep) or fractional (e.g. $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{4}$ of cell at each timestep).

Observations

Observations are representations that code the most relevant information of the environment. Basically, the Flatland package provides 3 kinds of observations:

- **Global:** it gives a global observation of the entire rail environment.
- **Local:** it gives a local observation of the rail environment around the agent. It is deprecated.
- **Tree:** it gives a local observation which exploits the graph structure of the rail network.



The way with which they are generated and encoded is one of the most important aspects of the challenge, because a good observation means more effectiveness during the training of a model and efficiency during computation.

For this reason, a custom observation has been designed. In particular, the basic idea was to exploit the tree-shaped observation provided by the Flatland package, since it is already highly efficient and, nonetheless, the amount of information is huge. Below is an example:

```
Direction  root :  0 ,  0 ,  0 ,  0 ,  0 ,  0 ,  33.0 ,  0 ,  0 ,  29 ,  1.0 ,  0
        Direction  L : -np.inf
        Direction  F :  inf ,  inf ,  inf ,  inf ,  inf ,  1 ,  32.0 ,  0 ,  0 ,  0 ,  1.0 ,  0
                Direction  L : -np.inf
                Direction  F :  inf ,  inf ,  inf ,  inf ,  2 ,  30 ,  3.0 ,  0 ,  0 ,  0 ,  1.0 ,  0
                Direction  R :  inf ,  inf ,  inf ,  inf ,  2 ,  32 ,  3.0 ,  0 ,  0 ,  0 ,  1.0 ,  0
                Direction  B : -np.inf
        Direction  R : -np.inf
        Direction  B : -np.inf
```

So, the focus was to generate a more compact observation, since the main problem encountered during all the experiments was the extreme slowness of the environment: even if the observations were transformed, they were too big and heavy to be handled smoothly.

To achieve compactness, a decomposition and encoding procedure of the observation of a single agent has been thought. In particular, it consists, first of all, in the transformation of the provided obs into a list of nodes. For each node, information about the direction to reach it and its depth, plus a series of twelve attributes has been stored by following a depth-first approach. In order to have a more compact representation all the values of the attributes have been added up, specifying a weight for each of them. In all those nodes in which a '-np.inf' value was present, a '-12' was inserted in substitution as a representative value (like it was the sum of twelve '-1'). However, the main problem encountered at this point was to maintain a fixed size for the decomposed observation (which depends actually by the depth of three). To overcome the problem, the decomposition has been completed by inserting in the empty part a representative value, that is '-1'. This is the final result for what regards the list of nodes:

```
('L', 1, -12.0, [('L', 2, -1), [...], ('F', 2, -1), [...], ('R', 2, -1), [...], ('B', 2, -1), [...]])
('F', 1, 13.600000000000001, [('L', 2, -12.0), ('F', 2, 15.100000000000001), ('R', 2, 12.900000000000002), ('B', 2, -12.0)])
('R', 1, 11.600000000000001, [('L', 2, -12.0), ('F', 2, 10.900000000000002), ('R', 2, -12.0), ('B', 2, -12.0)])
('B', 1, -12.0, [('L', 2, -1), [...], ('F', 2, -1), [...], ('R', 2, -1), [...], ('B', 2, -1), [...]])
```

Once the list of nodes is obtained, the next step is simply to stack all the information contained in the list, taking the encoding part for the directions of the nodes. At the end, an array of 63 elements for a single agent is obtained if depth=2 is considered:

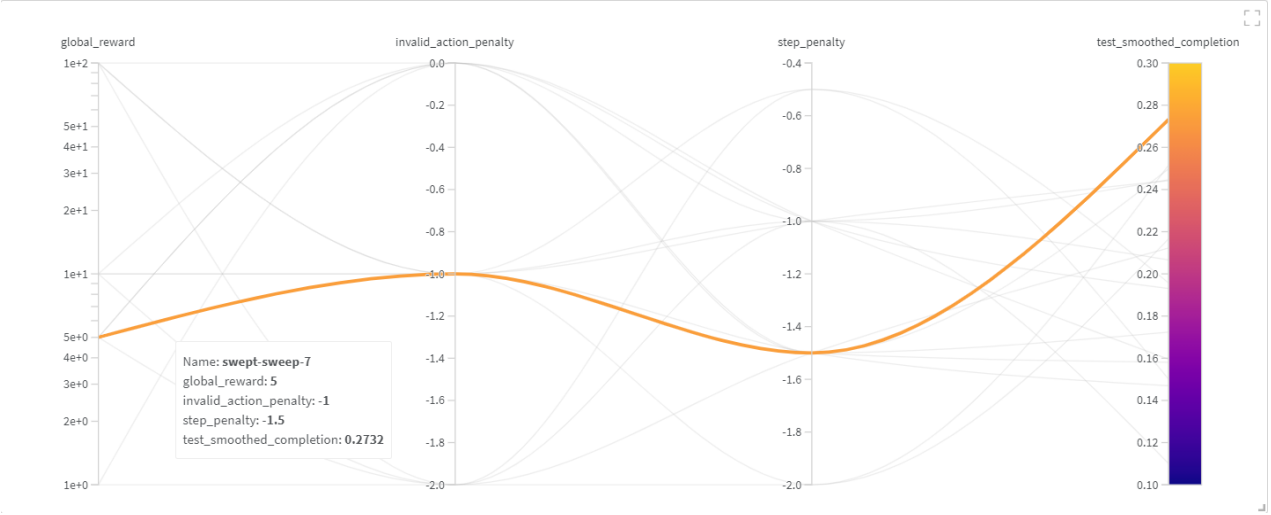
```
[  0.    0.   15.3   1.    1.  -12.    1.    2.   -1.    2.    2.   -1.
   3.    2.   -1.    4.    2.   -1.    2.    1.   13.6   1.    2.  -12.
   2.    2.   15.1   3.    2.   12.9   4.    2.  -12.    3.    1.   11.6
   1.    2.  -12.    2.    2.   10.9   3.    2.  -12.    4.    2.  -12.
   4.    1.  -12.    1.    2.   -1.    2.    2.   -1.    3.    2.   -1.
   4.    2.   -1. ]
```

Rewards

Rewards in Flatland are parametrized through two values α and β both set to 1.0 as default value. The possible rewards are:

- **invalid_action_penalty** = 0: it is awarded whenever an agent commits an invalid action. Disabled by default.
- **step_penalty** = $(-1 \times \alpha)$: it is awarded whenever an agent takes a step.
- **stop_penalty** = 0: it is awarded whenever an agent stops another agent. Disabled by default.
- **start_penalty** = 0: it is awarded whenever an agent sets a stationary agent in motion. Disabled by default.
- **global_reward** = $(1 \times \beta)$: it is awarded whenever an agent has reached its destination.

Since it is allowed to modify the above rewards, different ranges of values have been tested in order to achieve the best set of rewards and penalties. For example, it turns out that the best score for the 'global_reward' is 5; the best one for the 'invalid_action_penalty' is -1 while for the 'step_penalty' is -1.5. These are the selected quantities that maximize the total completion percentage.

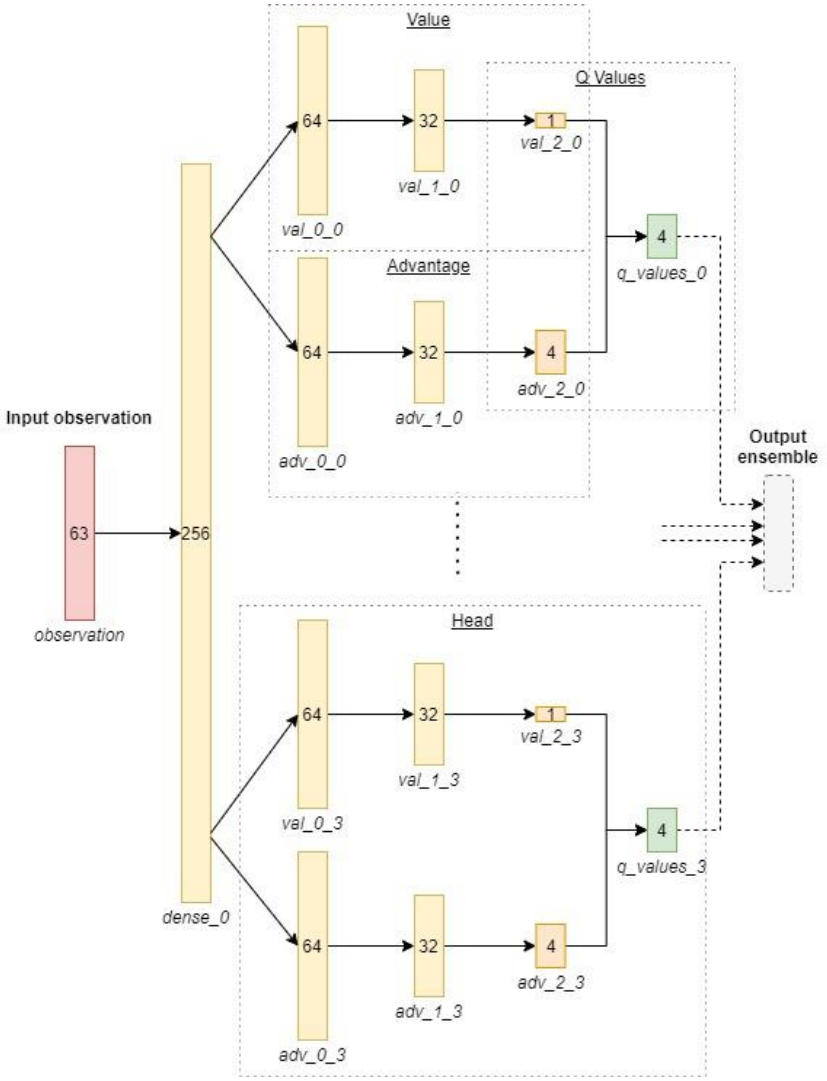


The model

Before proceeding by introducing the built model for the challenge, it is worth to specify that, in order to obtain the best hyperparameters, the *Weight & Biases* platform has been used. Also, it is very useful since it provides a lot of interesting graphs that can be used to understand where the model should be enhanced or simply for a presentation.

The network

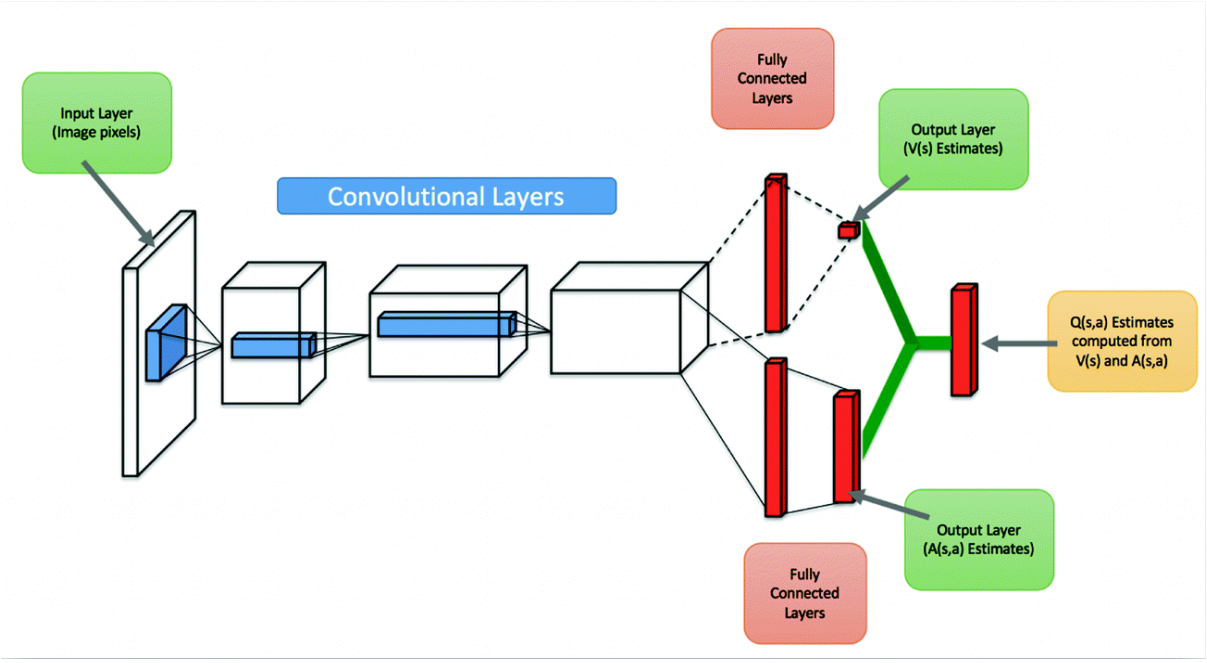
The network architecture that has been selected as a solution is the Dueling Double DQN which is the state of the art of Value-based Reinforcement learning techniques. This kind of network has been designed to solve a variety of problems and games such as Doom Deathmatch.



The complete structure of the network is the one above. The input is represented by an observation of an agent, as discussed before. This input gets preprocessed with a dense layer. Here there are multiple heads, with a dueling structure, that compute the resulting Q value. The network is trained with a Double approach.

Dueling DQN

In the Flatland's framework, the network is not characterized by the 3 convolutional layers as it normally is, because the input of the NN is not an image, but a flattened vector which is obtained by the observation of the environment. This choice reduces the complexity of the model as the useful information is already extracted. Below is a diagram of a dueling DQN network:



So, how does it work? Basically, the Dueling part means that the NN is split in 2 parts: the first computes an estimation of the value of the given state; the second computes the advantage of each action in order to understand which states are the most promising without exploring the outcomes of each possible action. At the end, values and advantages are combined with merging layers to obtain Q values.

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

Double DQN

The Double part means that, during training, 2 distinctive neural networks are used: one is the primary network which is responsible to compute the Q values used for the optimal action; the other is the target network which is used to estimate the Q value of the chosen action in the given state. The main advantage of this component is that it solves the problem of the overestimation of future rewards caused by the basic algorithm of the DQN.

$$G_t^{\text{DQN}} = R_{t+1} + \gamma Q(S_{t+1}, \underbrace{\underbrace{\operatorname{argmax}_a Q(S_{t+1}, a; \theta_t)}_{DQN \text{ Network choose action}}; \theta_t}_{\text{Evaluation of the } Q\text{-value by taking that action at that state}})$$

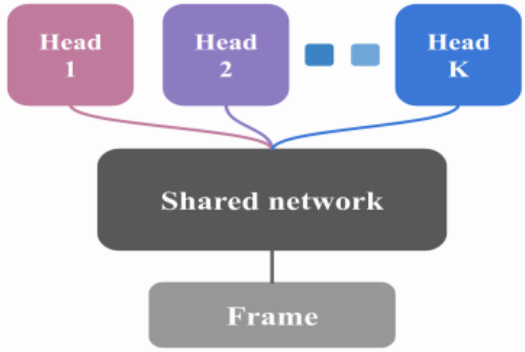
$$G_t^{\text{DoubleDQN}} = R_{t+1} + \gamma Q(S_{t+1}, \operatorname{argmax}_a Q(S_{t+1}, a; \theta_t); \theta'_t)$$

The exploiting/exploring component

For what concerns the exploiting/exploration part, which is responsible to fix a trade-off between the will of the algorithm to explore and the idea of exploiting the knowledge acquired up to a certain point during training, two possibilities have been tested:

- **ε-greedy:** it defines a certain probability ε with which the network is forced to choose a random action that enables it to deeper explore the environment. So, the agent explores a lot at the beginning, but then, since this probability decreases, it explores less, exploiting more the acquired knowledge.
- **bootstrapped DQN**: it trains K bootstrap heads (structural replicas of the last layers of the network), each on different slices of data: in every episode, a random head is selected to determine the next action. This is the uncertainty component which determines the exploration of the environment: each head has its own value function so the choice of a head can be seen from the point of view of the others as an exploration choice, something that they would never do. Each step in the replay buffer is enriched with a Bernoullian mask whose length is the same as the number of heads. This mask represents which heads will be trained with that experience, each with a fixed probability p.

During the test phase, then, to determine the next action, each head votes for its own best action. Thus, the action with the highest number of votes is selected. This reduces the use of multiple heads to an ensemble method. It happens that the more important the decision points are, the more the heads agree, and the other way around.

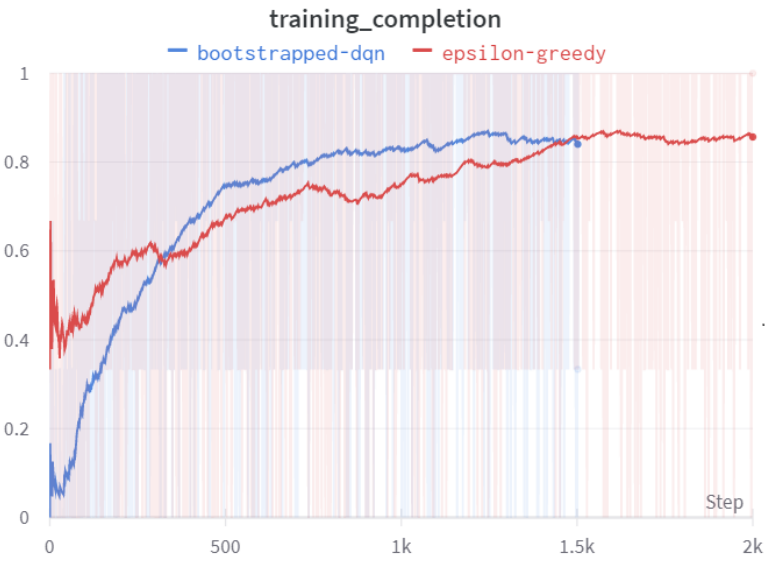


The bootstrapped DQN approach was originally designed with the aim of deeply exploring the environment in order to compensate for the demand for larger amounts of data.

So, for this part there are 2 hyperparameters: K, the number of heads; p, the probability of a head to be trained with an experience. The choice made is to stick K to the number of outcomes, 4, and a balanced probability p of 0.5.

As a consequence, this approach allows you to get good results with few iterations since there's always an exploration component during all the training but it is not random. The advantages are reported in the following section.

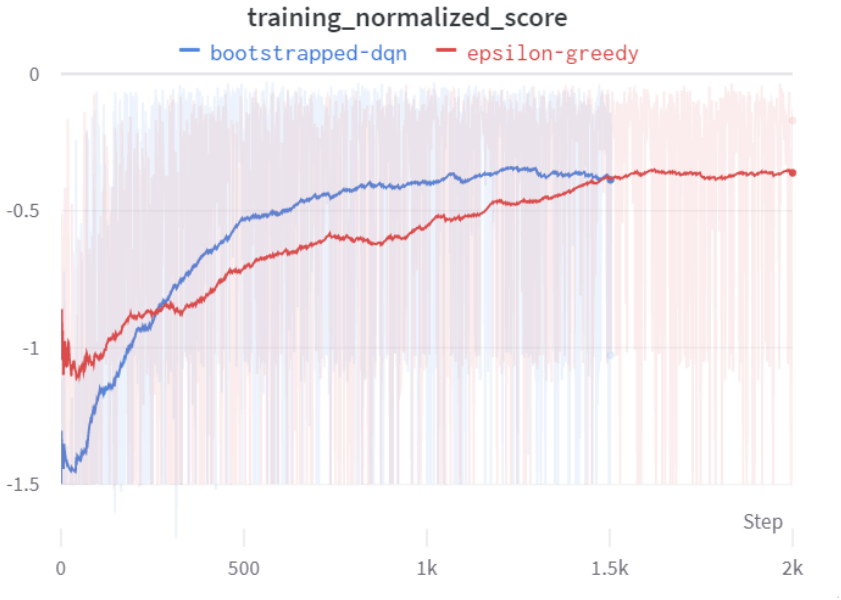
The results



What is immediately possible to deduce from the graph, which depicts the percentage of agents that have completed their tasks during the training phase, is how fast is the bootstrapped approach with respect to the more well-known ε-greedy approach. This latter, in particular, reaches the value of 0.8 around the 1300th iteration; with the bootstrapped DQN, the same score is reached around the 760th iteration with a saving almost ⅖ of the number of iterations.

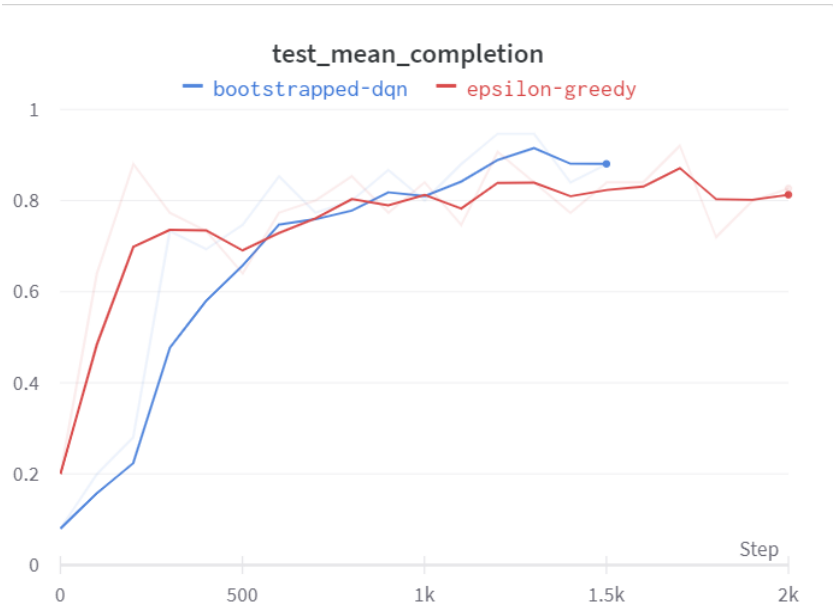
Even if the final value which they roam is identical, the speed and the consequent number of iterations that are saved lead the bootstrapped DQN approach to be preferred.

There are confirmations of the goodness of this approach also in the following graph which is about the scores during the training phase:

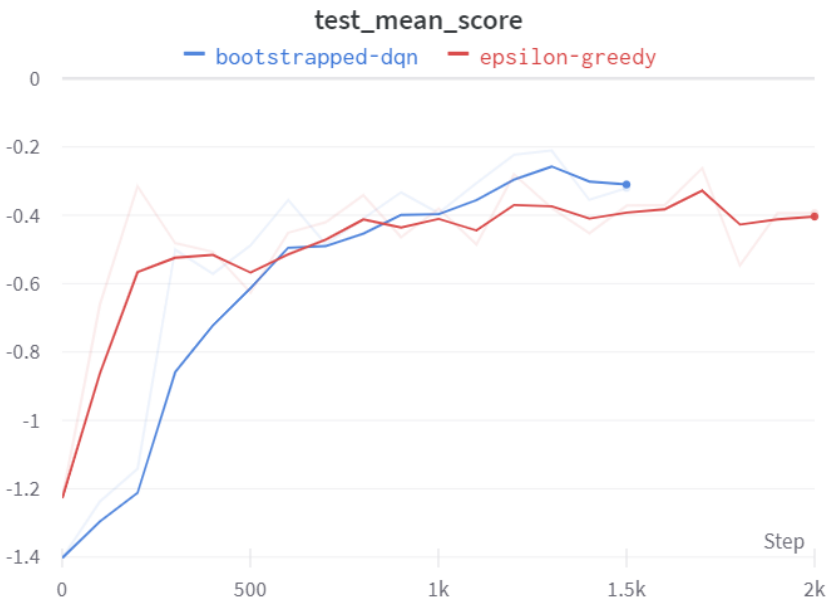


Also in this case, the bootstrapped DQN technique is way faster: the score of -0.5 is reached with a saving of almost one half (0.47) of the number of iterations.

For what regards the testing phase, there's an interesting pattern that is possible to observe in the next graph. In this case, that is the completion during the testing phase, in fact, the bootstrapped is not faster than the ε-greedy. Nonetheless, there's a point in which the first surpasses the second and potentially, observing the last part of the line (around the 1500th iteration), it would increase even more.



The same trend as before can be observed here below in this last graph which represents the mean score during the testing phase:



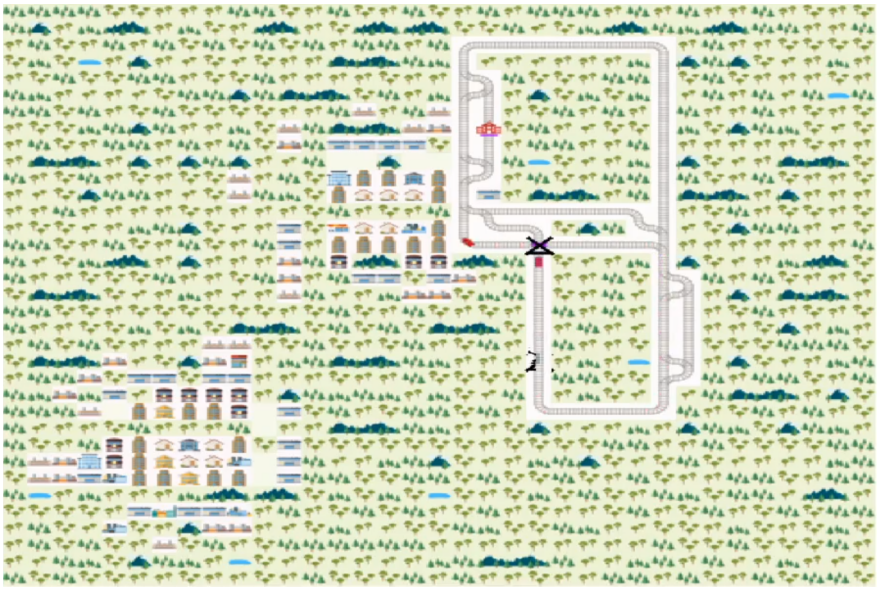
An additional graphical test has been run just to visualize the comparison between the two approaches.

Mean results on 25 test episodes					
Epsilon Greedy			Bootstrapped DQN		
Score	Done	Steps	Score	Done	Steps
-0,329	74,70%	258,68	-0,212	88,00%	165,12

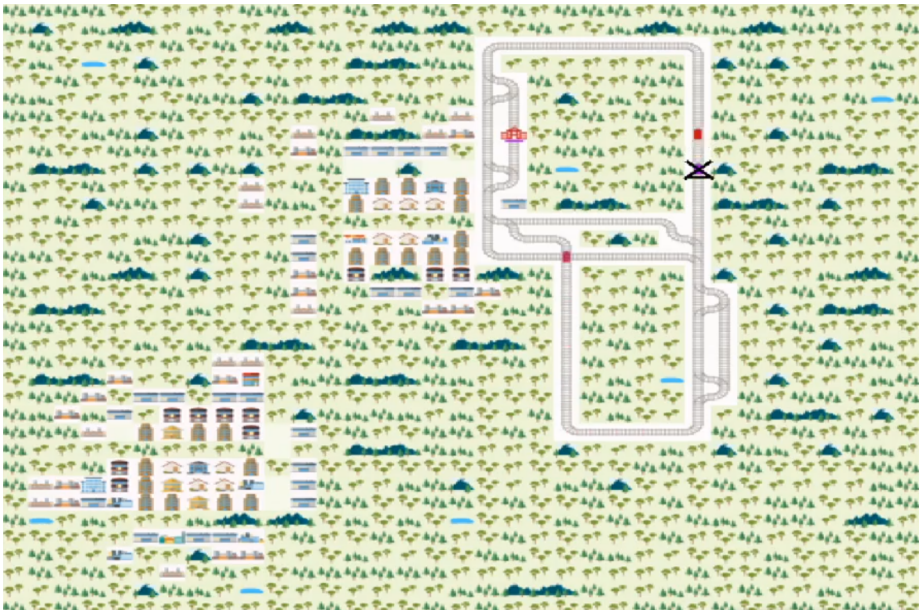
The table below shows the episodes in which the two approaches get different results. Most of the time it is the Bootstrapped DQN approach that outperforms the ε-greedy one. Unexpectedly, something interesting also has appeared: it regards how the two approaches manage malfunctions and local deadlock cases, of course in the same episodes.

	Epsilon Greedy			Bootstrapped DQN		
Seed	Score	Done	Steps	Score	Done	Steps
3	-0.679	33.3%	570	-0.099	100.0%	129
7	-0.704	33.3%	570	-0.093	100.0%	68
10	-0.116	100.0%	72	-0.142	100.0%	116
18	-0.695	33.3%	570	-0.089	100.0%	79
20	-0.689	33.3%	570	-0.109	100.0%	88
21	-0.703	33.3%	570	-0.182	100.0%	136
23	-0.706	33.3%	570	-0.091	100.0%	70
24	-0.142	100.0%	102	-0.724	33.3%	570

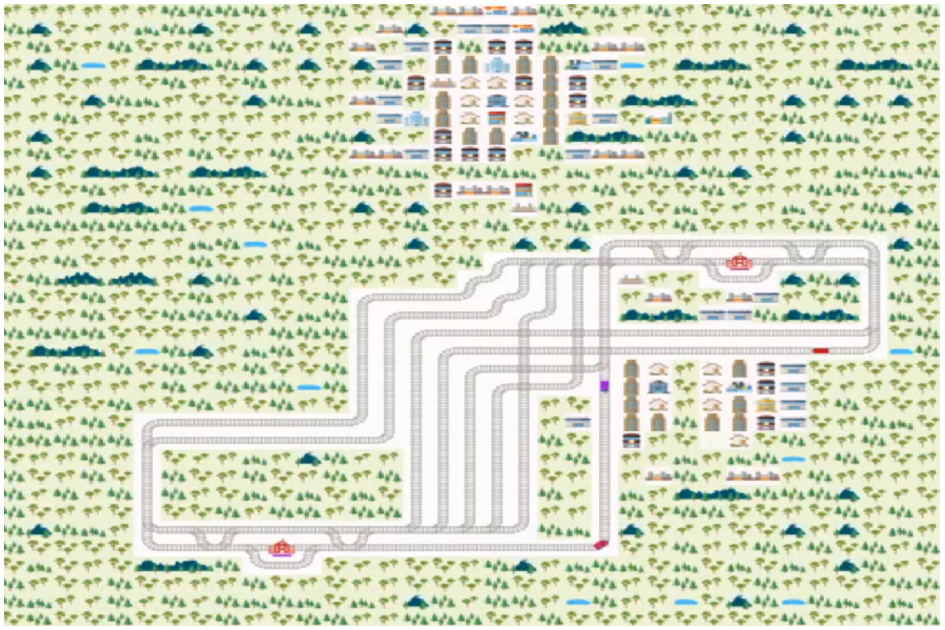
In the 10th case, despite the fact that the ε-greedy gets a better result, using fewer steps than the Bootstrapped DQN, the solution found by this latter should be considered the best.
Down below, in fact, it is possible to observe the ε-greedy case in which the higher score is achieved because the shortest path for each agent has been selected. However, this selection of actions leads the lower red train to stop due to the malfunctioning purple agent.



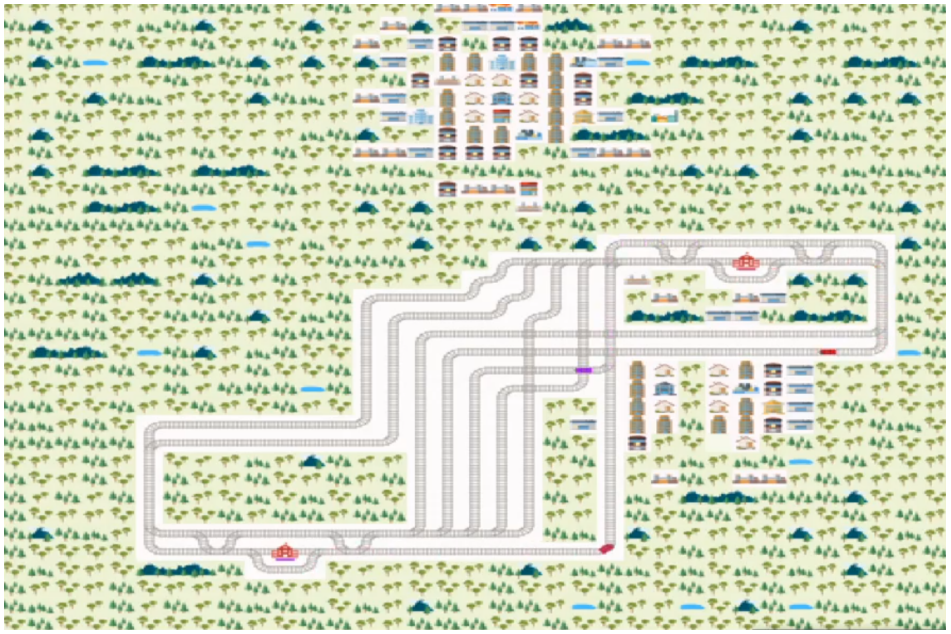
In the bootstrapped DQN case here below, instead, the model decides to choose another rail that leads to a longer route for some agent, but the management of the traffic is better, because there are less intersections between the trajectories of the agents.



Here is the case in which the ε-greedy fails at avoiding a local deadlock between two agents (the purple one and the fuchsia one).



On the contrary, the bootstrapped DQN has found a way to avoid the two agents to be stopped, by determining an alternative rail for the purple agent.



The conclusions

All the above examples have been demonstrated that the bootstrapped DQN technique is better, under certain perspectives, than the ε-greedy technique. The main reason is that the way the bootstrapped DQN reasons about exploring is not randomly determined, but, as it has been explained, is characterized by a certain distribution of uncertainty, unlike the other method which simply decreases. It's very promising and it is worth it to be studied, implemented and thorough.

This is what can be deduced, also, in general, about reinforcement learning: it's a field which is at its beginning, so something has yet to be filed, but it is worth pursuing especially in this kind of contexts.

Concerning the challenge, quite satisfying results can be considered as achieved, despite the model being simple and characterized by many approximations and inaccuracies. There were efficiency problems, these have been solved through custom observations and alternative techniques.

However, possible improvements can alway be conquered: a wider network with a higher number of layers, nodes; more tests for hyperparameters; new techniques which use images of the environment to determine the next best action, and so on and so forth.

References

Flatland package version 2.2.2 documentation.
(url = https://flatlandrl-docs.aicrowd.com/01_readme.html)

Hado van Hasselt et al., Deep Reinforcement Learning with Double Q-learning, CoRR, 2015. (url = http://arxiv.org/abs/1509.06461)

Ziyu Wang et al., *Dueling Network Architectures for Deep Reinforcement Learning*, CoRR, 2015. (url = http://arxiv.org/abs/1511.06581)

Ian Osband et al., *Deep Exploration via Bootstrapped DQN*, CoRR ,2016.
(url = http://arxiv.org/abs/1602.04621)
(yt = https://www.youtube.com/watch?v=6SAdmG3zAMg&ab_channel=IanOsband)