Flatland Challenge

G. Berselli, R. De Matteo, M. M. L. Pulici

July 19, 2021

Contents

List of Figures

1 Flatland Challenge [1]

The aim of the challenge, is to achieve efficient management of railway traffic. In particular, this problem is tackled in a simple grid world environment, in order to simulate and experiment different scenarios.

In more detail, the goal consists in making a number of trains arrive at their destinations minimizing the travel time. Even though for simple environments the train can follow explicit plans, as complexity increases the problem of mapping all possible states becomes intractable, for this reason a class of algorithms known as Reinforcement Learning (Section 2) can be exploited.

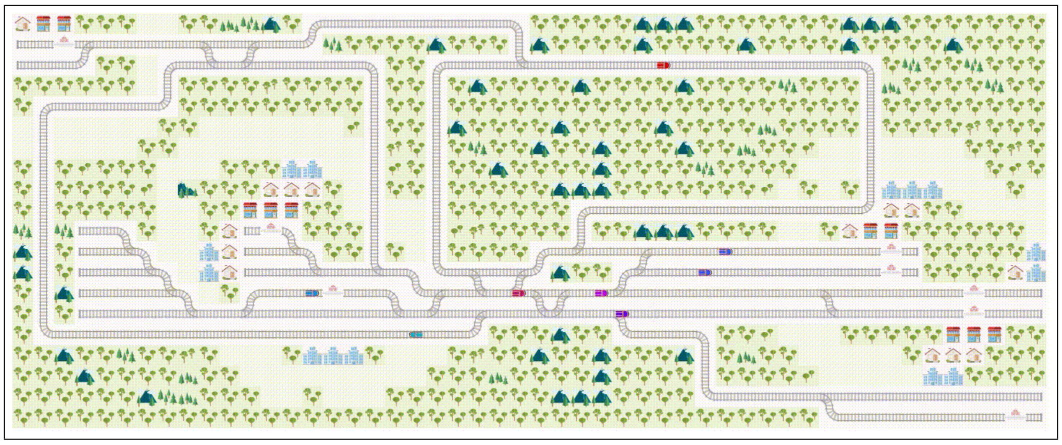

Figure 1: A possible Flatland instance

1.1 The environment [2]

The Flatland environment consists in a discrete time simulation, meaning that each action is performed with a constant time step. At each step, an agent for each simulated train chooses an action. An agent is defined as an entity that can move within the grid and must solve tasks. More precisely, each agent can choose between two actions: waiting or moving in a direction. Each agent has an individual starting position and its goal is to reach its target destination. Of course, two agents can not occupy the same cell at the same time.

Each cell in the Flatland grid can take the form of any of 8 tile types, as shown in Fig. 2. More configurations can be obtained by rotating and mirroring the 8 basic tiles.

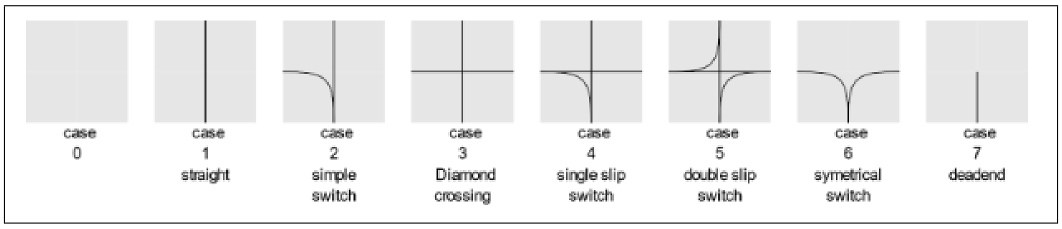

Figure 2: The 8 cell types

When the tile is of the straight type, the agent can only choose to continue moving or to stop. In presence of a simple or a double switch, the train is forced to decide in which of the offered direction to move. When arriving at a dead end, an agent can only stop or go backward.

1.2 Actions [3]

Since Flatland is a railway simulations, the agents' actions are naturally very limited. In particular, there are 5 basic actions:

- `DO_NOTHING`: if moving, the agent keeps moving, if stopped, it stays stopped;
- `MOVE_LEFT`: if the agent is at a switch with a leftward transition, it chooses the left path, otherwise it does nothing;
- `MOVE_FORWARD`: if the agent is at a switch with a straight transition, it goes straight, otherwise it does nothing;
- `MOVE_RIGHT`: if the agent is at a switch with a rightward transition, it chooses the right path, otherwise it does nothing;
- `STOP_MOVING`: the agent stops.

1.3 Observations [4]

The Flatland environment comes with 3 default observation: Global, Local Grid, and Local Tree. A graphical representation of the 3 observations is given in Fig. 3.

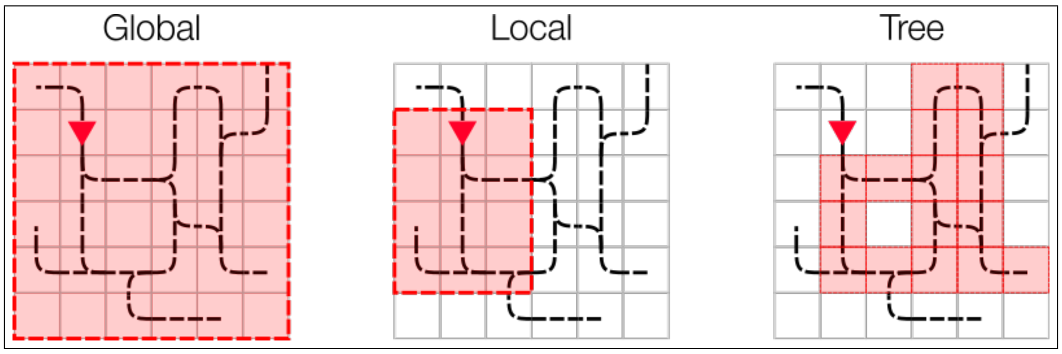

Figure 3: The 3 default Flatland observations

1.3.1 Global Observation

The Global Observations returns the entire rail environment. In more details, it is composed of:

- Transition map: provides a unique value for each type of transition map and its orientation;
- Agent state: a 3D array containing:
  - Channel 0: a one-hot encoding of the self agent's position and direction;
  - Channel 1: other agents' positions and directions;
  - Channel 2: self and other agents' malfunctions;
  - Channel 3: self and other agents' speeds;
  - Channel 4: the number of other agents ready to depart from that position;

- Agent targets: a 3D array containing the position of the self agent target and the positions of the other agents' ones.

1.3.2 Local Grid Observation

The Local Grid Observation returns the rail environment in a neighborhood of the agent. Its features are similar to that of the Global Observation, but the view field of the agent is limited. The main advantage over the Global Observation is that the Local Grid Observation reduces the amount of irrelevant information. In addition, it features a distance map, which gives information about the distance of each agent from its target.

1.3.3 Tree Observation

The Tree Observation exploits the graph structure of the railway. It generates a 4-branched tree starting from the agent's position, with each branch following an allowed transition. More information can be found in Section 3.4.1.

1.4 Rewards [3]

At each time step, each agent receives combined local and global rewards. At a local level, a reward $r_l = -1$ is given to each agent until its target location is reached. After this happens, the agent will receive a constant reward of $r_l = 0$. The global reward, on the other hand, by default starts at $r_g = 0$ and turns to $r_g = 1$ only after all agents have reached their targets. In total, the agent i receives a reward:

$$r_i\left(t\right) = r_l\left(t\right) + r_g\left(t\right). \tag{1.1}$$

1.5 Malfunctions [5]

Malfunctions are introduced in the Flatland environment in order to simulate stochastic events. In real life, the initial scheduling plan often needs to be changed because of unexpected events such as delayed arrivals or various other malfunctions. The way malfunctions are implemented is by using a Poisson process to stop agents at random times for random durations.

1.6 Speed profiles [6]

In a real railway, the difference in the various agents' speeds plays a huge role in increasing the complexity of the system. In Flatlands, this is simulated by using different speed profiles: a fastest speed is set to be 1, and all trains can take fractional speed values between 0 and 1.

2 Reinforcement Learning [7]

The term Reinforcement Learning refers to the class of agents which rely on feedbacks, or rewards, produced by the agents themselves when interacting with the environment. The way Reinforcement Learning works is to use observed rewards to learn the best possible policy for the given environment. In other words, the agent has no prior knowledge of the environment and must learn how to behave based only on posterior trial-and-error feedbacks.

There are three basic agent designs for Reinforcement Learning:

- Utility-based agent, which learns a utility function on states and uses it to select actions;
- Q-learning, which learns an action-utility function giving the expected utility of an action given a state;
- Reflex agent, which learns a policy mapping directly from states to actions.

In addition to these designs, Reinforcement Learning can be either passive, meaning that the policy is fixed and the task is to learn the utilities of states, or active, referring to agents which must also learn how to act.

2.1 Q-learning [8]

For the Flatland Challenge, Q-learning is used. This class of algorithms learns an action-utility representation instead of simply learning the utilities. The value of an action-state tuple is typically indicated as $Q\left(s,a\right)$. There is a direct correlation between Q-values and state utilities, as expressed by the following equation:

$$U\left(s\right) = \max_{a} Q\left(s,a\right). \tag{2.1}$$

The main difference between Q-functions and basic utility information si that no model of the form $P\left(s'|s,a\right)$ is needed, either for learning or for action selection. This characteristic makes Q-learning a model-free method.

The basic Q-learning algorithm is composed of the following steps:

1. Q-table initialization;
2. Action selection;
3. Q-table update.

A Q-table is a data structure which is used to keep track of the visited state-action pairs and the corresponding expected Q-values. At the very beginning of the learning process, the Q-table is filled with zeros, which represent the agent's lack of knowledge about the environment.

From here, the agent employs a trial-and-error strategy, choosing actions and balancing between exploration of new states and exploitation of previously collected knowledge. The way these actions are chosen depend on the so-called action selector, whose different types are explored in Section 3.2.

The third steps consists in updating the Q-table using the Bellman Equation:

$$Q\left(s,a\right) \leftarrow \left(1-\alpha\right) Q\left(s,a\right) + \alpha \left(r + \lambda \max_{a'} Q\left(s',a'\right)\right), \tag{2.2}$$

where s is the current state, a is the action taken, r is the reward for the taken action, α is the learning rate parameter and λ is the discount factor. The last max operation is performed on all actions a' which can be taken from the updated state s'.

A close alternative to standard Q-learning can be found in the Sarsa (State-Action-Reward-State-Action) algorithm. The concept is very close to that of Q-learning, but Eq. (2.2) is replaced by:

$$Q\left(s,a\right) \leftarrow \left(1-\alpha\right) Q\left(s,a\right) + \alpha \left(r + \lambda Q\left(s',a'\right)\right). \tag{2.3}$$

The only difference between Eqs. (2.2) and (2.3) is that $\max_{a'} Q\left(s',a'\right)$ is simply replaced by $Q\left(s',a'\right)$. In other words, instead of the maximum of the possible Q-values from the state reached in the transition, the actual value of the state s' reached after taking action a' is used. The value update takes place once at the end of each s, a, r, s', a' cycle, as suggested by the name.

If the agent is a greedy one, always choosing the action with the best Q-value, there is no difference between the two algorithms. If, on the other hand, there is some exploration happening, there is a crucial difference: Q-learning is said to be an off-policy algorithm, because it does not care about the policy used; SARSA, on the contrary, is considered on-policy. Even though Q-learning is more flexible, being able to learn well even with bad exploration policies, it is not as realistic, since it does not take into consideration possible external uncontrollable events.

2.2 Multi-Agent Reinforcement Learning [9]

Multi-Agent Reinforcement Learning (MARL) refers to a version of Reinforcement Learning where multiple agents interact in a common environment. There are three main broad categories which classify MARL:

- Cooperative: when all agents have a common goal and work together;
- Competitive: when agents compete to accomplish a goal;
- Mixed: when agents are grouped in teams, with intra-group cooperation and inter-group competition.

3 Implementation

3.1 Deep Q-Network Agents [8], [10], [11]

Learning using Deep Q-Networks (DQN) is a relatively new paradigm, having been introduced in 2014. In general, the main feature of DQN algorithms is the exploitation of Deep Neural Networks to learn the Q-values of a problem. There are many versions of DQN algorithms, of which the most common are:

- Standard DQN;
- Double DQN;
- Dueling DQN.

3.1.1 Standard DQN

The basic DQN algorithm closely resembles the standard Q-learning one, consisting of:

1. Neural network(s) initialization;
2. Action selection;
3. Network weights update.

The main difference between DQN and standard Q-learning is the way the Q-table is implemented. Rather than mapping the state-action pair to a Q-value, the neural network maps input states to action-value pairs.

Again, Eq. (2.2) is used to update the neural networks. In particular, the change of weights is expressed by:

$$\Delta w = \alpha \left[\left(r + \lambda \max_{a'} Q\left(s', a', w\right) \right) - Q\left(s, a, w\right) \right] \nabla_w Q\left(s, a, w\right), \tag{3.1}$$

where the part in the square brackets represent the temporal difference error and the last factor is the gradient of the current predicted Q-value.

3.1.2 Double DQN

When Eq. (2.2) is used in a Neural Network, the same parameters are used for estimating both the target and the Q-value. This is an issue, since it means that there is a big correlation between the target and the changing parameters: at every step, both the Q-values and the target values shift, leading to big oscillations in training.

A possible solution is to use the so-called Double DQN. The aim of this approach is to tackle the problem of checking that the best action for the next state is the one with the highest Q-value. In general, the accuracy of Q-values depend on both the action chosen and what states have been explored. Therefore, at the beginning of the training there is no information about the best action to take: taking the maximum Q-value can lead to false positives. This natural tendency of DQN to given higher Q-values to suboptimal actions complicates learning.

The Double DQN approach tries to fix this problem by using two separate networks to decouple the action selection from the target Q-value generation. First, the DQN network is used to select the best action to take for the next state, then the target network calculates the target Q-value of the state-action combination. So, the Double DQN method reduces Q-value overestimation and makes training faster and stabler. Practically, the time difference target $Q\left(s,a\right)$ becomes:

$$Q\left(s,a\right) = r + \lambda Q\left(s', \arg\max_{a'} Q\left(s',a'\right)\right). \tag{3.2}$$

Then, every T steps the weights w^{-} of the target network are updated with the DQN network ones w:

$$w^{-} \leftarrow w. \tag{3.3}$$

An alternative to this updating method of is given by the so-called soft update [12]. The term soft update refers to an approach which does not update the weights once every T steps, but rather frequently and very little every time. In particular, new weights are given by:

$$w^{-} = w\tau + w^{-}\left(1-\tau\right), \tag{3.4}$$

where τ is an arbitrary (small) parameter.

3.1.3 Dueling DQN

The third DQN version is based on decomposing the Q-value in two parts, as shown in Fig. 4:

$$Q\left(s,a\right) = V\left(s\right) + A\left(s,a\right) \tag{3.5}$$

where $V\left(s\right)$ is the value of being in state s and $A\left(s,a\right)$ is the advantage of taking action a at state s. To be more precise, the actual equation used is:

$$Q\left(s,a\right) = V\left(s\right) + A\left(s,a\right) - \frac{1}{|A|}\sum_{a} A\left(s,a\right), \tag{3.6}$$

where the last term is a normalization factor.

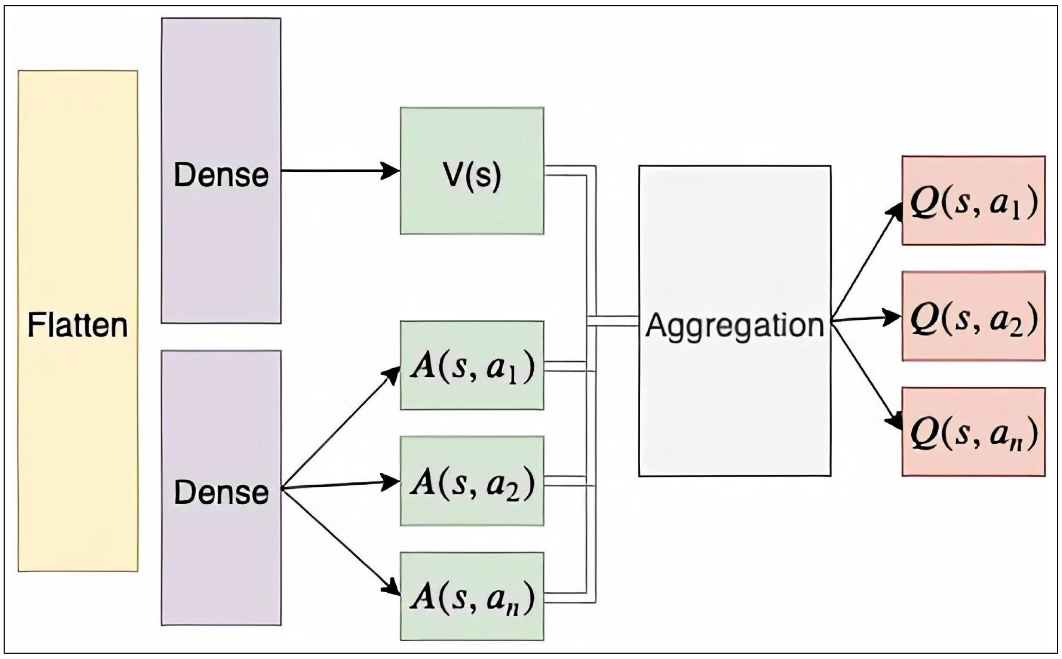

Figure 4: A schematic of the Dueling DQN

Dueling DQN focuses on separating the estimator of the two values, using one stream to estimate the state value $V(s)$ and another one to estimate the advantage of each action $A(s, a)$. Decoupling these two values becomes particularly useful for states where any action does not affect the environment in a relevant way.

3.1.4 Further DQN approaches

Of course, these three methods are not the only ones to deal with DQNs. A particularly exhaustive analysis of all the different improvements to the DQN algorithm is represented by the seminal work by Hessel et al. [13]: in the paper, different DQN approaches are compared, culminating in the creation of a state-of-the-art integrated "Rainbow" agent, exploiting all the advantages of the most successful approaches.

3.2 Action Selectors [14]

In the learning process of a Reinforcement learning agent, exploration plays a crucial role: in order for an agent to properly learn from the interaction with the environment, it must be exposed to as many states as possible. Since an agent needs the right experiences to learn a good policy, but also needs a good policy to obtain the environment, a balance, known as exploration-exploitation tradeoff, needs to be reached. There are various action selection approaches which can be used by the agent, of which the main ones are:

- Random Approach;
- Greedy Approach;
- ε-Greedy Approach;
- Boltzmann Approach;
- Noisy Approach.

3.2.1 Random Approach

The Random Approach is the most basic method of selecting an action: the agent does not use any prior knowledge and picks randomly among all the possible actions. This approach is illustrated in Fig. 5.

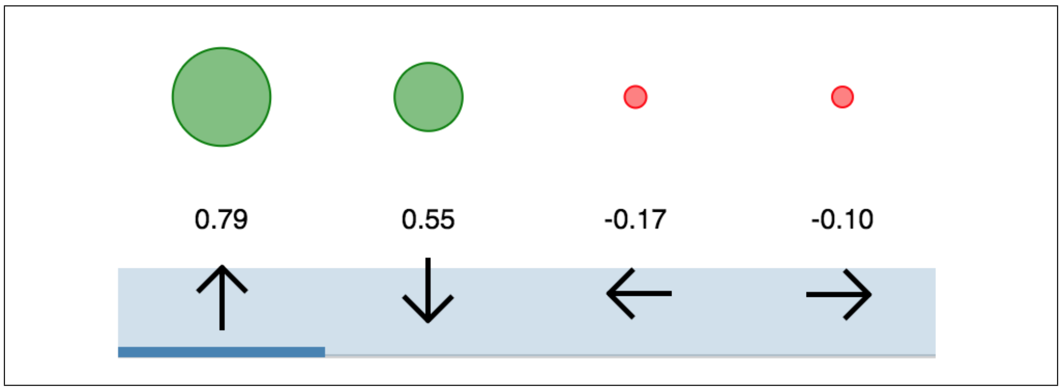

Figure 5: Action selection distribution for the Random Approach

Despite providing a lot of exploration, this approach is obviously deficient in exploiting the knowledge already learned by the agent.

3.2.2 Greedy Approach

The Greedy Approach is the extreme opposite of the Random one: it consists in always opting for the action with the highest Q-value, regardless of the values of the other choices. This approach is illustrated in Fig. 6.

At first, this approach might appear good, as the agent always opts for the action it thinks to be the best. The main shortcoming of this method is that it almost always provides a suboptimal solution, since no alternate solutions are explored. In other words, in Greedy Approach, exploitation is favored enormously over exploration, which is almost absent.

3.2.3 ε-Greedy Approach

The ε-Greedy Approach can be viewed as a combination of the Greedy and the Random ones. The way the agent acts in this case is by always opting

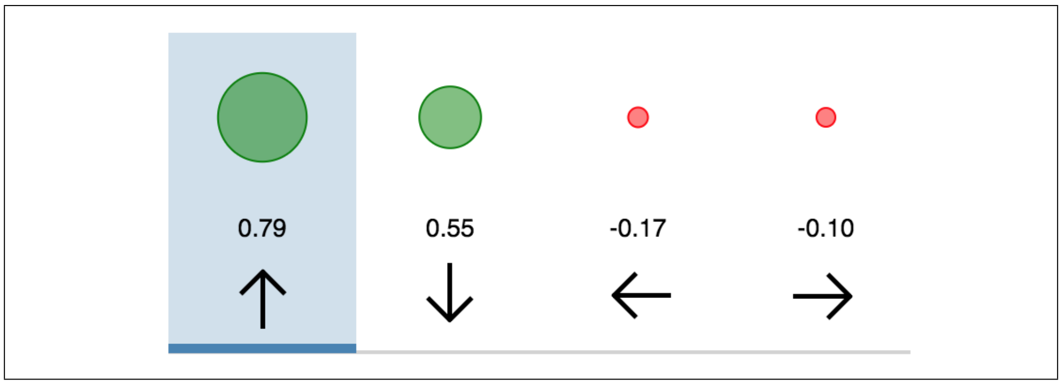

Figure 6: Action selection distribution for the Greedy Approach

for the optimal action, except occasionally it acts randomly. The choice between the two approaches is dictated by an adjustable parameter ε, which represents the probability to act randomly. This approach is illustrated in Fig. 7.

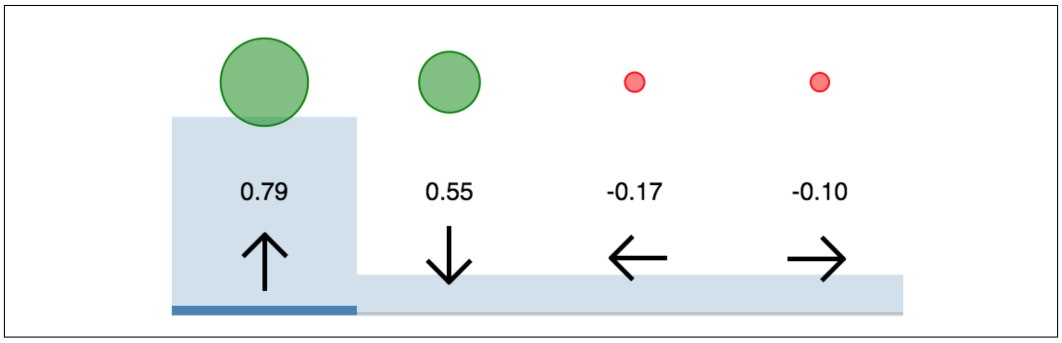

Figure 7: Action selection distribution for the ε-Greedy Approach

This approach encountered a huge success due to its combination of simplicity and power: even though it is only a mixture of two very mediocre methods, the performance improvement is remarkable. To further enhance the agent's learning ability, the value of ε is often adjusted during training: in the beginning it starts as a big value, in order to provide maximum exploration, and it is slowly annealed as the agent obtains more information about the environment. The only shortcoming of the ε-Greedy Approach is that it only takes into account whether an action is the most rewarding or not, making it not optimal.

3.2.4 Boltzmann Approach

The Boltzmann Approach takes the exploration-exploitation balance of ε-Greedy even further: instead of always taking the optimal action or acting randomly, it chooses among the various actions using individual Q-values to weigh probabilities. This approach is illustrated in Fig. 8.

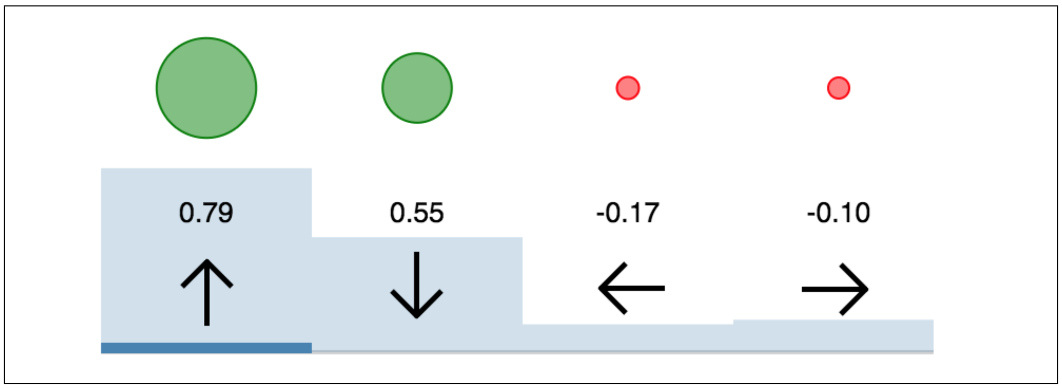

Figure 8: Action selection distribution for the Boltzmann Approach

Compared to ε-Greedy, the Boltzmann Approach also takes into consideration the information about the values of actions other than the optimal: this way, actions which are potentially promising are given a higher priority over clearly inferior choices.

An interesting feature of the Boltzmann Approach is the use of an additional temperature parameter τ, which is annealed over time in a fashion similar to the way ε is treated. The τ parameter controls the probability distribution of the actions using the thermodynamics Boltzmann equation, which gives the name to the approach:

$$P\left(a\right) = \frac{\mathrm{e}^{\frac{Q(a)}{\tau}}}{\sum_{i=1}^{n} \mathrm{e}^{\frac{Q(a_i)}{\tau}}}, \tag{3.7}$$

where $P\left(a\right)$ is the probability of choosing action a, and a_i are all the possible action choices.

The main problem of this approach is that it builds on the assumption that the probability distribution outlined in Eq. (3.7) provides a measure of the agent's confidence in action a, while in reality what the agent is estimating is a measure of how optimal the agent thinks the action is, not how certain it is about that optimality.

3.2.5 Noisy Approach [15], [16]

The Noisy Approach is somewhat different from all the others: instead of acting on the probability of the agent choices, it adds some noise to the output of the Neural Network itself. In particular, the basic Noisy Approach consists in adding some gaussian node to the last layers of the network using the formula:

$$w_i = \mu + \sigma \cdot \varepsilon \tag{3.8}$$

where μ is a variable with random initialization, σ is a variable with constant initialization, and ε is the noise with a random value between 0 and 1.

There are two main ways of introducing noise to the Network: Independent Gaussian Noise and Factorized Gaussian Noise. Using the first method, every weight of the noisy layer is independent, with its own μ and σ. The Factorized version, on the other hand, uses two vectors: the first one has the same length as the input, the second one the same as the output. A special function is then applied to both vectors and a matrix multiplication between them is calculated and is then used to add weights.

3.3 Experience Replay [17]

In basic Q-learning algorithms, experiences are utilized just once and then thrown away. This is a waste, as some experiences may be rare and some may be costly. Therefore, experiences should be reused effectively. An experience is defined as a quadruple (s, a, s', r), meaning that the execution of action a in a state s results in the state s' with reward r.

The most straightforward way to reuse experiences is the so-called Experience Replay, which consists in storing past experiences and then randomly sampling and presenting past experiences to the learning algorithm. This method results in a sped-up learning process, meaning that the networks usually converge more quickly.

3.3.1 Prioritized Experience Replay [18]

Prioritized Experience Replay is a type of experience replay which consists in giving priority to transitions with high expected learning progress, measured by their temporal-difference error: in other words, precedence is given to those transitions whose expected rewards differ the most from the experienced ones. This approach leads to two issues: firstly, the prioritization can produce a loss of diversity; secondly, it can introduce some bias.

The diversity loss problem can be solved by employing stochastic prioritization, a sampling method which interpolates between pure greedy prioritization and uniform random sampling: on the one hand, it ensures a monotonic sampling probability in a transition's priority; on the other hand, even the lower-priority transitions are guaranteed a non-zero probability. Practically, the probability of sampling a transition i is given by:

$$P\left(i\right) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \tag{3.9}$$

where p_i is the priority of transition i and α is the amount of prioritization which is used.

The introduction of a bias can be corrected by using importance-sampling weight, given by:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P\left(i\right)}\right)^\beta, \tag{3.10}$$

which fully compensates for the non uniform probabilities if $\beta = 1$. For stability reasons, weights are normalized by $\frac{1}{\max_i w_i}$, si that they only scale the update downwards.

3.4 Observation

Two observations are used in this project. The first one is the Tree Observation, one of the 3 default ones; the other one is a custom observation, the Density Observation.

3.4.1 Tree Observation [4]

As already anticipated, the Tree Observation consists in a 4-branched tree starting from the agent's position and exploring the graph structure of the railway, as shown in Fig. 9. Each node is composed of 12 features:

- Channel 0: if own target lies on the explored branch the current distance from the agent in number of cells is stored;
- Channel 1: if another agents target is detected the distance in number of cells from current agent position is stored;
- Channel 2: if another agent is detected the distance in number of cells from current agent position is stored;
- Channel 3: possible conflict detected;
- Channel 4: if a not usable switch is detected the distance is stored;
- Channel 5: distance to the next node;
- Channel 6: minimum remaining travel distance from node to the agent's target given the direction of the agent;
- Channel 7: number of agents going in the same direction found on path to node;
- Channel 8: number of agents going in the opposite direction found on path to node;
- Channel 9: if an agent has a malfunctioning, the number of time steps the observed agent will remain blocked is stored;

- Channel 10: slowest observed speed of an agent in same direction;
- Channel 11: number of agents ready to depart but no yet active.

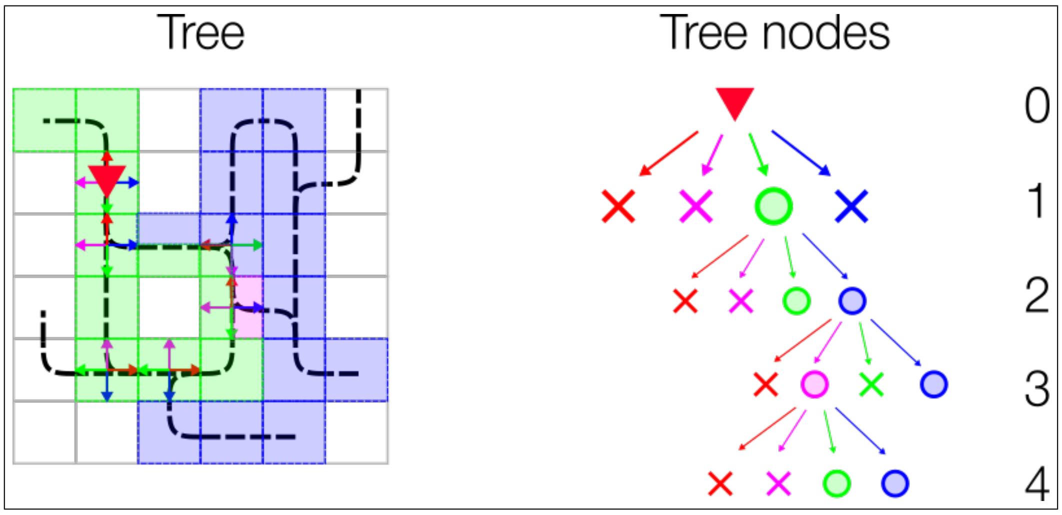

Figure 9: The Flatland Tree Observation

3.4.2 Density Observation

The custom Density Observation exploits Flatland's Shortest Path Predictor to assign each cell a value encoding information about if and when it will be occupied by a train. The values of all agents for a single cell are combined into a density value, depending on the time differences between the moments when the various agents occupy a given cell.

More specifically, each agent uses two matrices to learn: one matrix encodes the density map for the agent itself, while the other one is assigned density values depending on the other agent's predicted paths. Then the two matrices are stacked into a $w \times h \times 2$ matrix, where w and h are respectively the width and the height of the environment, which is then fed to a Convolutional Neural Network (CNN).The two matrices are stacked and not simply concatenated in order to preserve the spatial correspondency between the two matrices' density values. After 3 filtering layers of increasing size, the CNN output is flattened and passed to a standard Dueling DQN.

3.5 Proximal Policy Optimization [19], [20]

Proximal Policy Optimization (PPO) is a policy gradient-based method of reinforcement learning, designed to make some improvements on the Trust Region Policy Optimization (TRPO) algorithm, which in turn was designed to improve the Advantage Actor Critic (A2C) method. From the TRPO, the PPO inherits the importance sampling version of the policy loss function, which enables better sample efficiency, and which is expressed as follows:

$$L^{\mathrm{CPI}}(\theta) = \mathbb{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\mathrm{old}}}}A_t\right] = \mathbb{E}_t\left[r_t(\theta)A_t\right], \tag{3.11}$$

where the superscript CPI stands for "Conservative Policy Iteration". This loss function is an improvement of the one expressed by the A2C method. The policy loss indeed is no more subjected to the so-called "destructively large policy updates", typical of gradient-based methods, which maybe leads to update the parameters so far outside of the range, but now the policy gradient steps are limited, defining a ratio $r(\theta)$ between the action under the current policy and the action under the previous policy. PPO goes even further, introducing its main feature called "clipped surrogate objective", which can be expressed as follows:

$$L^{\mathrm{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)A_t, \mathrm{clip}\left(r_t(\theta), 1-\varepsilon, 1+\varepsilon\right)A_t\right)\right]. \tag{3.12}$$

The loss function is given by the expectation computed over a minimum of two terms: a normal objective and a clipped objective. The key component comes from the second term where a normal objective is truncated with a clipping operation between $1-\varepsilon$ and $1+\varepsilon$, where ε is a hyperparameter. Because of the min operator, the clipped objective behaves differently when the advantage estimate is positive or negative.

The effect of clipping is shown in Fig. 10. On the left situation, when the selected action has a better-than-expected effect, the loss function flattens out when the action is much more likely under the current policy compare to the old one. This is done to prevent overdoing an action update by taking a step too far. The same happens to the graph on the right: the loss function

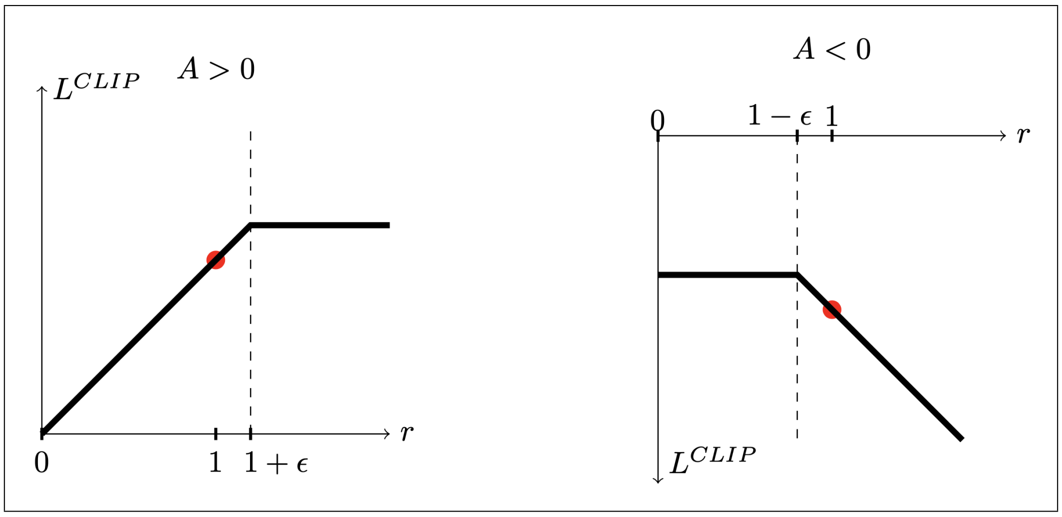

Figure 10: The L^{CLIP} function for positive advantages $A > 0$ and negative advantages $A < 0$; the red circles show the starting points for the optimization

flattens out when the action is much less likely under the current policy. This constraint is very powerful and avoid the computation of the additional KL constraint used in the TRPO in order to define a trust region, known to add overhead to optimization processes which sometimes leads to an undesirable training behavior. The total loss function for PPO then can be expressed as follows:

$$L_t^{\text{CLIP+VF+S}}(\theta) = \mathbb{E}_t \left[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}}(\theta) + c_2 S[\pi_\theta](s_t) \right], \tag{3.13}$$

where c_1 and c_2 are hyperparameters that weigh the various components of the loss, S denotes an entropy bonus (added to push the policy to behave more spontaneously until the other part of the objective starts dominating and so encouraging greater exploration), and L_t^{VF} is a squared-error loss (with the latter two terms identical to the components in A2C and with signs introduced to convert a maximization function to a minimization function).

4 Results

All training runs are performed with the following environment:

- 3 agents;
- Width $w = 30$;
- Height $h = 30$;
- 2 cities;
- Maximum 2 rails between cities;
- Maximum 2 rails inside each city.

Experiment tracking is performed using the Weights and Biases tool [21].

All graphs show the number of steps on the x axis and the average completion on the last 100 runs. The plots are smoothed with Weights and Biases's Gaussian filter with a power of 50.

4.1 Deep Q-Network agents

The plot shown in Fig. 11 represents 3 runs with different DQN agents and the following fixed settings:

- ε-Greedy action selector;
- Standard Experience Replay;
- Tree Observation.

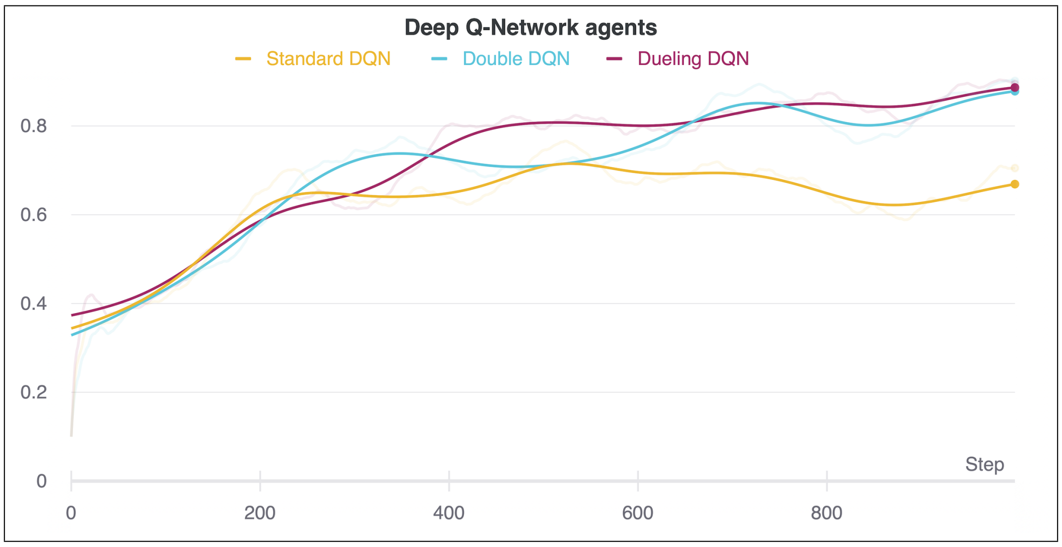

Figure 11: Deep Q-Network agents

The Standard DQN performs much worse compared to the Double and Dueling DQN agents. This is in line with what is expected according to the theory exposed in Section 3.1.

4.2 Action selectors

The plot shown in Fig. 12 represents 3 runs with different action selectors and the following fixed settings:

- Standard DQN agent;
- Standard Experience Replay;
- Tree Observation.

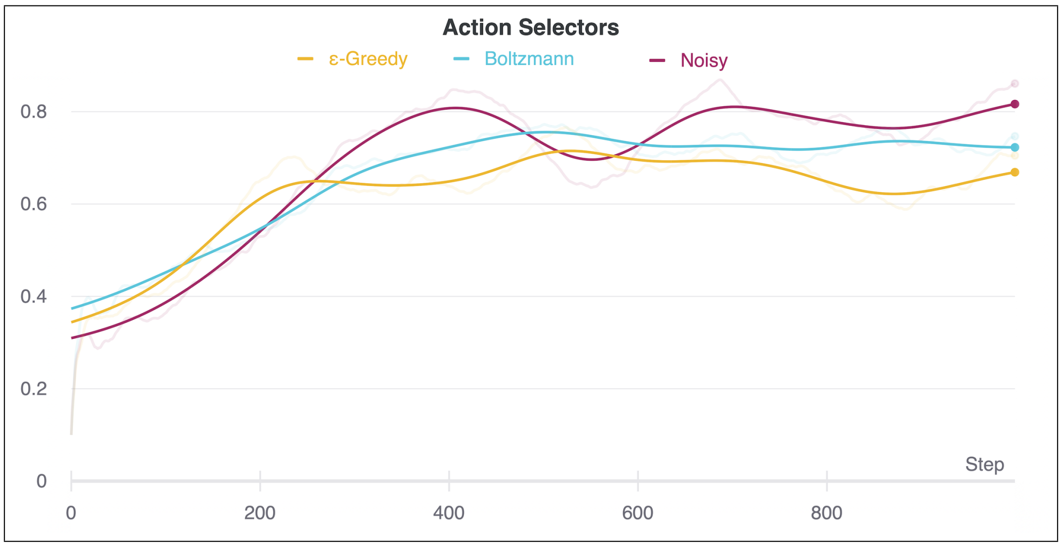

Figure 12: Action selectors

The Noisy action selector is clearly the best of the 3, with ε-Greedy and Boltzmann being quite close to each other. Again, these results are consistent with the expectations, given the content of Section 3.2.

4.3 Experience Replay

The plot shown in Fig. 13 represents 2 runs with different Experience Replay paradigms and the following fixed settings:

- Standard DQN agent;
- ε-Greedy action selector;
- Tree Observation.

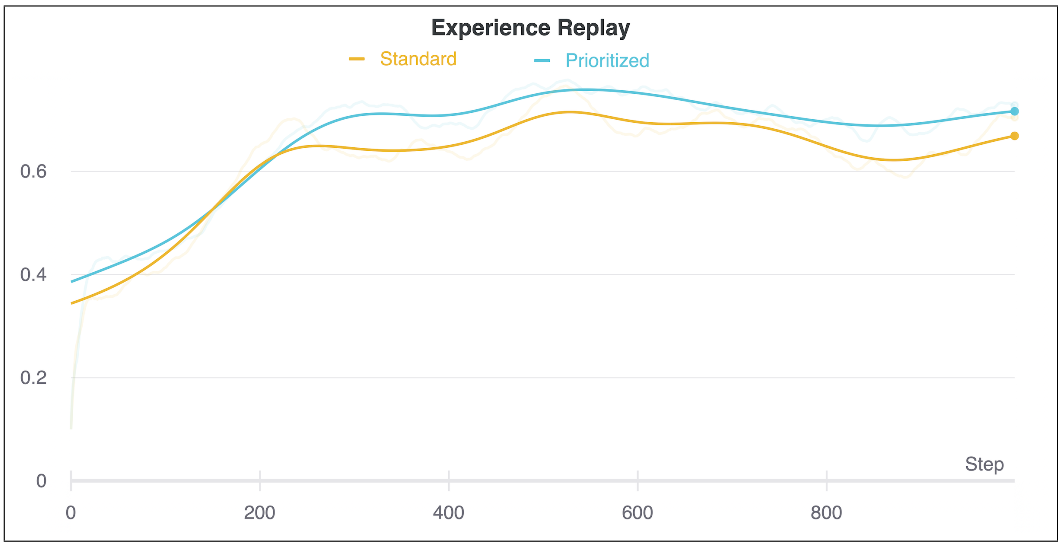

Figure 13: Experience Replay

The Prioritized Experience Replay run performs slightly better than the Standard one, in line with the theory of Section 3.3.

4.4 Observations

The plot shown in Fig. 14 represents 2 runs with different observations and the following fixed settings:

- Dueling DQN agent;
- ε-Greedy action selector;
- Standard Experience Replay.

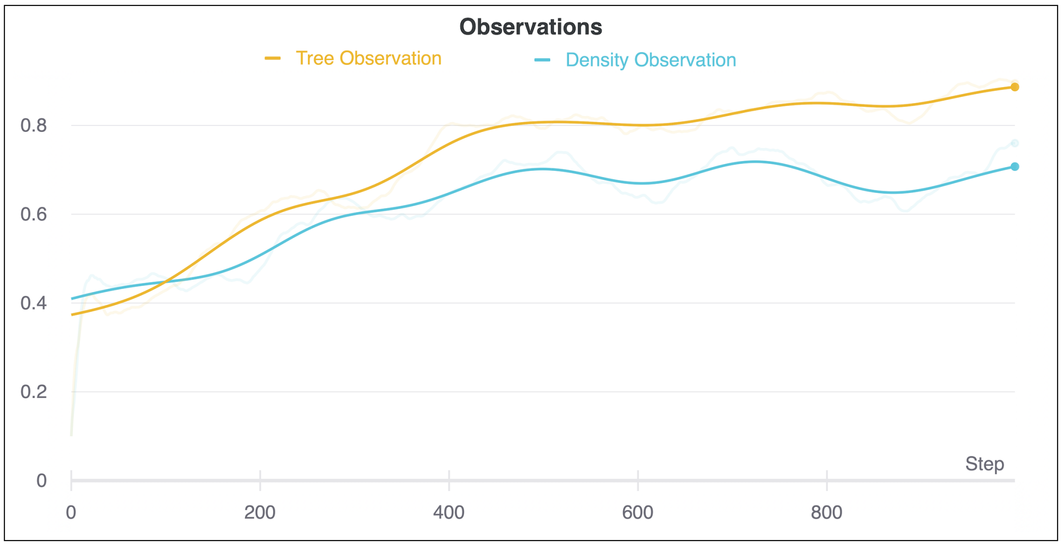

Figure 14: Observations

Even though it is a custom observation, Density Observation did not perform as bad as expected, despite being clearly worse than the Tree Observation run, which is also the reason why Tree Observation was preferred for all the other runs.

4.5 Proximal Policy Optimization

The plot shown in Fig. 15 represents 3 runs with different PPO parameters. The legend labels correspond to a triple of $\varepsilon/c_1/c_2$.

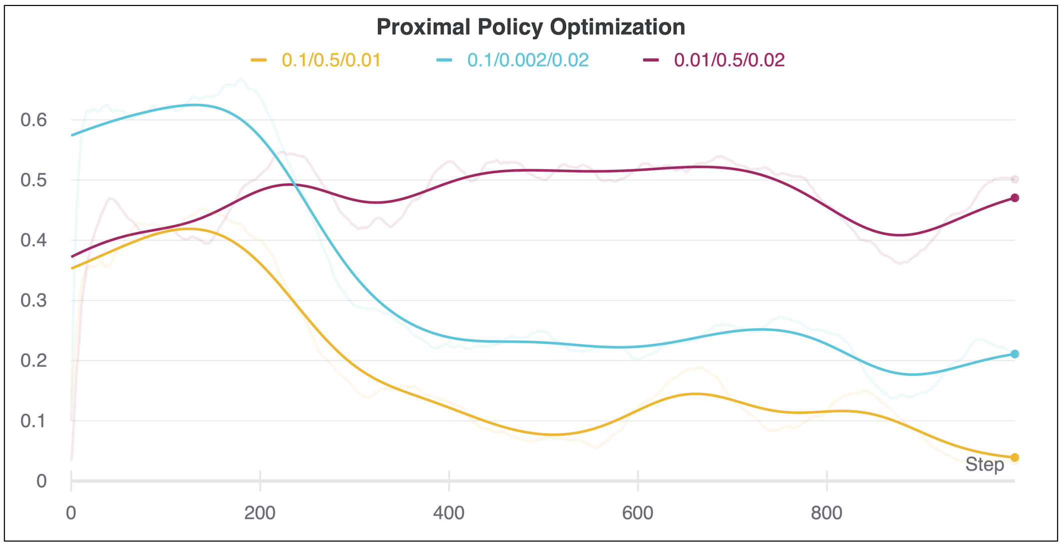

Figure 15: Proximal Policy Optimization

Using a bigger ε value (0.1), when the agent starts to go down a bad path it is unable to recover, resulting in a swift decrease of completion percentage. This behavior is particularly evident when using a small entropy weight (c_2). On the contrary, using a small ε value (0.01) and a big entropy weight (0.02) helps recovering after a bad path is taken, giving the best results. In any case, the performance of PPO methods is clearly worse than the runs shown in previous plots.

References

[1] S. Mohanty, E. Nygren, F. Laurent, M. Schneider, C. Scheller, N. Bhattacharya, J. Watson, A. Egli, C. Eichenberger, C. Baumberger, G. Vienken, I. Sturm, G. Sartoretti, and G. Spigler, “Flatland-RL : Multi-Agent Reinforcement Learning on Trains”, AIcrowd, 2020. arXiv: `2012.05893 [cs.LG]`.

[2] (2020). Flatland Challenge, AIcrowd, [Online]. Available: `https://www.aicrowd.com/challenges/flatland-challenge` (visited on 07/19/2021).

[3] (2020). Flatland Environment, The Flatland Community, [Online]. Available: `https://flatland.aicrowd.com/getting-started/env.html` (visited on 07/19/2021).

[4] (2020). Provided Observations, The Flatland Community, [Online]. Available: `https://flatland.aicrowd.com/getting-started/env/observations.html` (visited on 07/19/2021).

[5] (2020). Malfunctions, The Flatland Community, [Online]. Available: `https://flatland.aicrowd.com/getting-started/env/stochasticity.html` (visited on 07/19/2021).

[6] (2020). Speed profiles, The Flatland Community, [Online]. Available: `https://flatland.aicrowd.com/getting-started/env/speed_profiles.html` (visited on 07/19/2021).

[7] S. J. Russell and P. Norvig, “Reinforcement Learning”, in *Artificial Intelligence, A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Pearson Education, 2009, ch. 21, pp. 830–859.

[8] M. Wang. (2020). Deep Q-Learning Tutorial: minDQN, Medium, [Online]. Available: `https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc` (visited on 07/19/2021).

[9] P. Haou. (2021). Multi-Agent Reinforcement Learning (MARL) and Cooperative AI, Medium, [Online]. Available: `https://towardsdatascience.com/ive-been-thinking-about-multi-agent-reinforcement-learning-marl-and-you-probably-should-be-too-8f1e241606ac` (visited on 07/19/2021).

[10] T. Simonini. (2018). Improvements in Deep Q Learning: Dueling Double DQN, Prioritized Experience Replay, and fixed..., freeCodeCamp, [Online]. Available: `https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/` (visited on 07/19/2021).

[11] M. Sewak, "Deep Q Network (DQN), Double DQN, and Dueling DQN", in *Deep Reinforcement Learning, Frontiers of Artificial Intelligence*. Singapore, Singapore: Springer Nature Singapore, 2019, ch. 8, pp. 95–108.

[12] (2019). Learn Reinforcement Learning (3) - DQN improvement and Deep SARSA, greentec.github.io, [Online]. Available: `https://greentec.github.io/reinforcement-learning-third-en/` (visited on 07/19/2021).

[13] M. Hessel, W. Dabney, J. Modayil, D. Horgan, H. van Hasselt, B. Piot, T. Schaul, M. Azar, G. Ostrovski, and D. Silver, "Rainbow: Combining Improvements in Deep Reinforcement Learning", in *Proc. 32nd AAAI Conference on Artificial Intelligence*, (New Orleans, Louisiana, USA, Feb. 2–7, 2018), pp. 3215–3222. [Online]. Available: `https://aaai.org/ocs/index.php/AAAI/AAAI18/paper/download/17204/16680`.

[14] A. Juliani. (2016). Simple Reinforcement Learning with Tensorflow Part 7: Action-Selection Strategies for Exploration, Medium, [Online]. Available: `https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-7-action-selection-strategies-for-exploration-d3a97b7cceaf` (visited on 07/19/2021).

[15] P. H. Moghadam. (2019). Deep Reinforcement learning: DQN, Double DQN, Dueling DQN, Noisy DQN and DQN with Prioritized Experience Replay, Medium, [Online]. Available: `https://medium.com/@parsa_`

h_m/deep-reinforcement-learning-dqn-double-dqn-dueling-dqn-noisy-dqn-and-dqn-with-prioritized-551f621a9823 (visited on 07/19/2021).

[16] M. Fortunato, M. G. Azar, B. Piot, J. Menick, M. Hessel, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg, "Noisy Networks for Exploration", presented at the 6th International Conference on Learning Representations (Vancouver, Canada, Apr. 30–May 3, 2018). arXiv: 1706.10295 [cs.LG].

[17] L.-J. Lin, "Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching", *Machine Learning*, vol. 8, pp. 293–321, May 1992. DOI: 10.1007/BF00992699.

[18] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay", presented at the 4th International Conference on Learning Representations (San Juan, Puerto Rico, May 2–4, 2016). arXiv: 1511.05952 [cs.LG].

[19] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms", OpenAI, Aug. 28, 2017. arXiv: 1707.06347 [cs.LG].

[20] T. Kim. (2021). Understanding Proximal Policy Optimization (Schulman et al., 2017), Medium, [Online]. Available: https://towardsdatascience.com/understanding-and-implementing-proximal-policy-optimization-schulman-et-al-2017-9523078521ce (visited on 07/19/2021).

[21] *Weights and Biases*, Weights and Biases, 2021. [Online]. Available: https://wandb.ai/site (visited on 07/19/2021).