ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Master in Artificial Intelligence - Deep Learning course

Flatland Project

Marco Cucè, Alessandro Stockman, Alessandra Stramiglio

Academic Year 2020/2021

Contents

List of Figures

1 Introduction

Flatland [1] is a challenge organized by AIcrowd in collaboration with the Swiss Federal Railways (SBB) aimed at managing dense traffic on complex railway networks in an efficient way.

It represents a real-world problem faced by many transportation and logistics companies. The goal of the challenge is to plan the path of an arbitrary number of trains inside a rail environment by guiding them towards a target station with minimal travel time by minimizing the number of steps that it takes for each agent to reach its destination.

The purpose of this work is to find a solution using modern Reinforcement-Learning approaches.

1.1 Reinforcement Learning

Reinforcement Learning (RL) is one of the three main Machine Learning methods (Supervised Learning, Unsupervised Learning, Reinforcement Learning). RL lies in between full supervision and complete lack of predefined labels. On one hand, it uses well-known methods of supervised learning, such as **deep neural networks** for function approximation, stochastic gradient descent and backpropagation to learn data representation. On the other hand, RL applies those methods in a different way. The key components of RL are the **Agent**, the **Environment** and **Rewards**.

- The **reward** is a scalar value obtained periodically from the environment representing the quality of the agent's behaviour. The frequency of the reward is not predefined, but it's common practice to set it to a fixed timestamp or at every environment interaction. The term *reinforcement* derives, indeed, from the fact that reward obtained by an agent should reinforce its behaviour in a positive or negative way. Reward is *local*, it reflects the success of the agent's recent activity and not all the successes achieved by the agent so far.

- The **agent** is somebody or something that interacts with the environment by executing certain actions based on reasoning on observations and therefore receiving rewards for this. In practical scenarios, the agent is a portion of software that is supposed to solve some problem in an ideally efficient way.

- The **environment** represent everything outside of an agent. The agent's communication with the environment is limited to reward (obtained from the environment), actions (executed by the agent) and observations (information the agent receives about the environment).

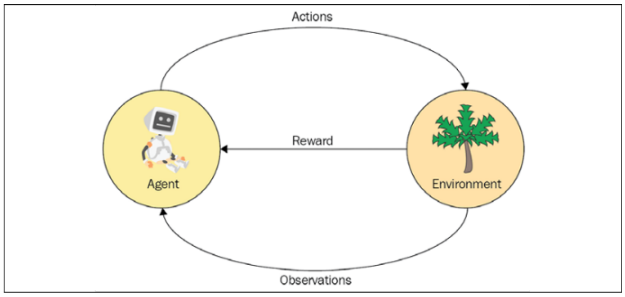

Figure 1: Graphical representation of RL entities and their communication channels.

The agent performs **actions** in the environment which can be either discrete or continuous: the former being a set of mutually exclusive interactions that an agent can perform and the latter involving continuous values.

The state of an environment differs from the agent's view by including every detail about the environment itself, while the observation represents what the agent perceives about it. A temporal sequence of environment's states is usually called *episode*.

1.1.1 Markov Decision Process

Markov Decision Process (MDP) is used to formulate any problem in RL mathematically. In the **Markov Process Chain**, X_t represents the state of the process at time t, with $t \in \mathbb{Z}^+$. Each move from state i to j in one time-step has a *transition probability* p_{ij}. This probability is fixed, i.e. it does not change over time.

This process defined above has the property that the probability of moving from some state i to another state j is independent of the states visited prior to i.

$$P(X_{t+1} = j | X_t = i, X_{t-1} = i_{t-1}, ..., X_1 = i_1, X_0 = i_0) = P(X_{t+1} = j | X_t = i) = p_{ij}$$

The transition probability can be captured by a transition matrix, which is a NxN matrix, where N is the number of states in our model and the row index i represents the source state and the column j represents the target state and each cell is equivalent to the probability to go from the source state to the target state.

In Reinforcement learning, the goal is maximizing the cumulative reward (all the rewards agent receives from the environment) instead of, the reward agent receives from the current state (also called immediate reward). Adding reward to the Markov Chain leads to the **Markov Reward Process**. Each reward is multiplied by a discount factor γ in order to weight the importance of the immediate and future rewards. This total sum of reward the agent receives from the environment, is called *return* and is defined as follows.

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

It follows that a Markov Reward Process is a Markov chain with values judgement. Basically, a value is obtained from every state our agent is in. Mathematically, a Markov Reward Process is defined as $R_s = \mathbb{E}[R_{t+1}|S_t]$.

The last step to obtain the MDP is to add agent's actions. The Markov Process and Markov Reward Process had transition matrix, able to express the probability to jump from a source state to a target state. Hence, the agent is able to choose an action to take at every state transition, the 2D matrix becomes 3D matrix where the depth is governed by the possible actions the agent can perform to reach the target state from the source state.

1.1.2 Policy

The policy is the core concept in which Reinforcement Learning is built around, it is defined by the set of rules controlling the agent's behaviour. The main objective of RL is to obtain as much **return** as possible and different policies can provide different amounts of return, therefore, the ideal goal is to find the *optimal policy* for the agent.

Formally, policy is defined as the probability distribution over actions for every possible state:

$$\pi(a|s) = P[A_t = a|S_t = s]$$

1.1.3 Bellman Equation

Optimal policies are really hard to formulate and it's even harder to prove their optimality. Intuitively a policy could contain "greedy" features. Bellman Equation is used to find optimal policies and value function. The policy changes with experience therefore, the value function will differ based on the policies. Optimal value function is one which gives maximum value compared to all other value functions.

The value of the state is $V(s) = \mathbb{E}[\sum_{t=0}^{\infty} r_t \gamma^t]$, where r_t is the local reward obtained at a step t of the episode.

The total reward could be discounted with γ or not (undiscounted case equals to $\gamma = 0$). The value is always calculated in terms of some policy that the agent follows.

To introduce this concept it's better to start by putting ourselves into a deterministic case, where all actions have a 100% guaranteed outcome. Assume that the agent observes state s_0 and has N available actions, every action leads to another state $s_1, ..., s_N$ with respective reward, $r_1, ..., r_N$. Also, assume that the values V_i of all states connected to state s_0 are known. The best course of action that the agent can take in such state is:

$$V_0 = max_{a \in 1...N}(r_a + \gamma V_a)$$

In this equation also the long-term value of the state is considered. Bellman proved that with this extension, the behaviour will get the best possible outcome, in other words, it will be optimal. The above equation is called the Bellman equation of value.

In addition to the value of the state, *V(s)*, it's convenient to introduce the value of the action, *Q(s,a)*. It represent the total reward the agent can get by executing action a in state s and can be defined via *V(s)*. Even if it is a less fundamental entity than *V(s)*, this quantity gave a name to a whole family of methods called **Q-learning**. In

this methods, our primary objective is to get values for Q for every pair of state and action.

V(s) can also be written via *Q(s,a)*:

$$V(s,a) = max_{a \in A} Q(s,a)$$

This means that the value of some state equals to the value of the maximum action the agent can execute from this state. *Q(s,a)* results as follows:

$$Q(s,a) = r(s,a) + \gamma max_{a' \in A} Q(s',a')$$

Q-values are much more convenient in practice, as for the agent, it's much simpler to make decisions about actions based on Q than on V. In the case of Q, to choose the action based on the state, the agent just need to calculate Q for all available actions using the current state and choose the action with the largest value of Q. To do the same using values of the states, the agent needs to know not only values, but also probabilities for transitions. In practice they are rarely known in advance, so the agent needs to estimate transition probabilities for every action and state pair.

1.1.4 Multi-Agent Reinforcement Learning (MARL)

The multi-agent case is often reflected into real cases, where several agents are involved in the environment interaction. Of course, those agents can communicate between each other, in terms of communication approaches it's possible to define two kind of agents:

- Competitive: When two or more agents try to beat each other in order to maximize their reward.
- Collaborative: When a group of agents need to use joint efforts to reach the goal.

In multi-agent case, most of the elements are dependent on all the agents:

- The state transitions are the result of the joint action of all the agents.
- The rewards of the agents depend on the joint action.
- Their returns depend on the joint policy.
- The Q-function of each agent depends on the joint action and on the joint policy.

In fully cooperative stochastic problems, the reward functions are the same for all the agents. It follows that the returns are also the same and all the agents have the same goal: to maximize the common return.

On one hand, MARL (cooperative) can lead to a faster learning and better performance, it is possible to have a speed-up thanks to parallel computation when the agents exploit the decentralized structure of the task. Furthermore, by design most multi-agent systems also allow the easy insertion of new agents into the system, leading to a high degree of scalability. On the other hand, we have the issue of the curse

of dimensionality that is caused by the exponential growth of the discrete state-action space, because basic RL algorithms like Q-learning estimate values for each possible discrete state or state-action pair, this growth leads directly to an exponential increase of their computational complexity. Of course, MARL's complexity is also exponential in the number of agents, since each agent adds its own variables to the joint state-action space.

1.2 Flatland-RL Library

AICrowd provides a framework which implements a Reinforcement Learning environment for the Flatland challenge, which comes with several components attached to the library:

- Actions:

  The trains in flatland have five possible actions to execute at each step:

  - DO_NOTHING: Repeats the action of the previous step (if the agent is already moving it continues moving, if it is stopped, it remains still). In the case of an agent in a a dead-end, this action will result in the train turning around.
  - MOVE_LEFT: Valid only at cells where the agent can change direction towards the left. If chosen, a rotation of the agent orientation to the left is performed. If the agent is stopped, this action will cause it to start moving in any cell where forward or left is allowed.
  - MOVE_FORWARD: Forward movement. At switches will chose the forward direction.
  - MOVE_RIGHT: Analogue to MOVE_LEFT.
  - STOP_MOVING: This action causes the agent to stop.

- Observations:

  Three default observations are provided by the framework, it is possible, though, to provide custom ones:

  - Global Grid Observation
  - Local Grid Observation
  - Tree Observation

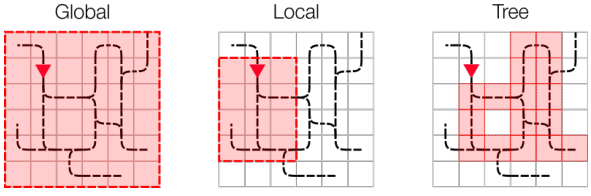

Figure 2: Visual summary of the three provided observation.

- Rewards:

  At each time step, each agent receives a combined reward which consists of a **local** and a **global** reward signal. Locally, the agent receives $r_l = -1$ for each time step, and $r_l = 0$ for each time step after it has reached its target. The global reward signal r_g is equal to 0 only if all agents have reached their targets, in the opposite case it is worth 1. Every $i - th$ agent receives:

  $$r_i(t) = \alpha r_l(t) + \beta r_g(t)$$

  α and β are two tuning factors to regulate collaborative behaviour. In the 2021 Flatland challenge the provided values are $\alpha = 1.0$ and $\beta = 1.0$

1.2.1 Malfunctions

Malfunctions are a complication added to the Flatland problem to simulate delays by stopping agents at random times for random duration. This random value is implemented using a Poisson process. Trains that malfunction can't move for a random, but known, number of steps and they block the trains following them.

2 Solution

2.1 Preliminary Analysis

To start experimenting on the framework, we initiated a small environment to test the improvements of the training algorithms on the *completions* percentage and average *scores* over the episodes.

Environment Parameters:

Parameter	Value
n_agents	3
x_dim	30
y_dim	30
n_cities	2
max_rails_between_cities	2
max_rails_in_city	2
min_malfunction_interval	10000
grid_mode	False
malfunction_duration	[20, 50]

2.1.1 Observation

After some tests the most effective provided observation has been noticed to be the *TreeObservation*. This observation helps to exploit the fact that a railway network

is a graph and thus, the observation is only built along allowed transitions in the graph. The observation is generated by spanning a 4 branched tree from the current position of the agent. Each branch follows the allowed transitions (backward branch only allowed at dead-ends) until a cell with multiple allowed transitions is reached. Here the information gathered along the branch is stored as a node in the tree.

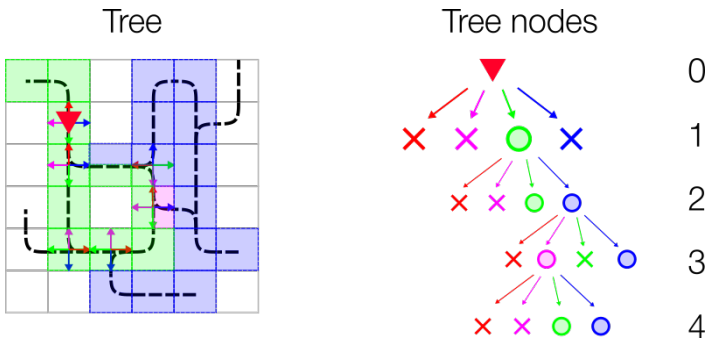

Figure 3: Tree observation schema.

2.1.2 Stochastic Path Predictor

A further help given by the *Flatland RL* library comes from the implementation of a standard **Predictor**, which may be implemented in an observation to allow the agents to exploit an approximation of the other agent's future positions, helping to prevent deadlocks.

Specifically, this concept is applied in the 4^{th} channel of the *TreeObservation*. The default predictor, with its basic implementation, assumes every agent to travel through their shortest path.

However, this approach results in a simplistic behaviour, since the agents won't always travel along that way. Given the presence of multiple agents and the limited number of tracks, the task of the policy should be to put the largest number of agents in the condition to reach the target avoiding deadlocks. This leads agents to choose also longer path in order to try not to block other agents.

To overcome this simplification, we decided to implement a random factor to the predicted position by implementing a predictor able to guess the agents' future position by sampling a probability distribution proportional to the distance between the possible paths and the agent's target.

Given a distance D_{fp} between the future position fp and the agent's target and the sum T of distances of all the available paths, the probability of that position to be chosen is given by the following random variable FP:

$$P(FP = fp) = \frac{T - D_{fp}}{2 \cdot T} \tag{1}$$

2.1.3 Deadlock Detection

Since the agents sometimes may present competitive behaviour, it can happen that they lead to a deadlock situation. This happens when an agent cannot perform any action because it is blocked by one or more agents. In a full deadlock situation, no agent would arrive to its own target. This conflict situation is quite common and it increases in frequency proportionally to the number of agents and the dimension of the grid decreasing. A good point to start with, is to track the deadlocks happening in an episode. In order to do so, for each agent in the environment we check if its *RailAgentStatus* is "ACTIVE" and if the position in which it has to go is feasible or it is occupied by another agent. By checking if each agent is in deadlock, we create a list of booleans indicating whether the agent is deadlock or not. The data collected is then logged and used to have more insights on the quality of the policy.

2.2 Policy Gradients

The policy can be written as $\pi(s) = argmax_a Q(s,a)$ which means that the result of our policy π, at every state s, is the action with the largest Q-value. The policy is what to look for when solving RL problem. Indeed, when the agent obtains the observation and needs to make a decision, it needs the policy. In the case where the request is to obtain Q-values, the policy is represented by a neural network that returns values of actions as scalars.

To be able to obtain those values, the gradient on the policy is computed, which is defined as:

$$\nabla J \approx \mathbb{E}[Q(s,a)\nabla \log \pi(a|s)] \tag{2}$$

It follows that the idea is trying to increase the probability of actions that have given good total reward and decrease the probability of actions with bad final outcomes. On practical view point, policy gradient can be implemented by performing optimization of the following loss function:

$$\mathcal{L} = -Q(s,a)log\pi(a|s)$$

During SGD the quantity is minimized, therefore there is the minus sign in front of the equation.

The general algorithm of policy gradient methods is synthesized in:

1. Initialize the network with random weights;
2. Play N full episodes, saving their *(s,a,r,s')*;
3. For every step,t, of every episode, k, compute the discount total reward for subsequent steps:
$$Q_{k,t} = \sum_{i=0} \gamma^i r_i$$

4. Calculate the loss function for all transitions:

$$\mathcal{L} = -\sum_{k,t} Q_{k,t} log(\pi(s_{k,t}, a_{k,t}))$$

5. Perform an SGD update of weights, minimizing the loss;

6. Repeat from step 2 until converged.

This preceding algorithm is different from Q-learning one in several aspects: no explicit exploration, no replay buffer, no target network used. One of the way to improve policy-gradient method, is to reduce the variance of the gradient:

$$var[x] = \mathbb{E}[(x - \mathbb{E}[x])^2]$$

If there is a value which has a large variance it could slow down our training significantly cause it may require many samples to *average out* his shift effect. To overcome this problem, it is convenient to subtract the mean total reward from the Q-value, this mean is called **baseline**.

Actor-Critic methods goes further in reducing variance by making the baseline state-dependent, since different states could have very different baselines. The total reward can be represented as a value of the state plus the advantage of the action: $Q(s, a) = V(s) + A(s, a)$. So, using *V(s)* as baseline is not a bad idea, the problem is that *V(s)* is not known. It's possible to use another neural network that will approximate *V(s)* for every observation. To train, it's possible to use the same training procedure that is used in DQN methods. When the value (or an approximation) for any state is known, it's possible to use it in order to calculate the policy gradient and update the policy network such that probabilities for actions with good advantage values are increased. The policy network is called *Actor*, since it tells what to do. The other network is called *Critic*, as it allows to understand how good the action were. This improvement is known under the name *Advantage Actor-Critic* (A2C).

2.2.1 Proximal Policy Optimization (PPO)

The core improvement over the basic A2C method is to tweak around the formula used to estimate the policy gradient. The PPO method [2] uses as objective function the ratio between the new and the old policy, scaled by the advantages:

$$J_\theta = E_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_old}(a_t|s_t)} A_t]$$

To limit the update of the above policy gradient function, it is clipped. Let's write the ratio between the new and the old policy as:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_old}(a_t|s_t)}$$

Then the clipped objective can be formulated as:

$$J_\theta^{clip} = \mathbb{E}_t[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)]$$

This new objective limits the ratio between the old and the new policy in the interval $[1-\epsilon, 1+\epsilon]$, so by varying ϵ, size of the update is tuned.

Another substantial difference from A2C method is how the advantage estimation is computed. In classic A2C it's:

$$A_t = -V(s_t) + r_t + \gamma r_{t+1} + ... + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_t)$$

where T are the steps.

In PPO there is a more general estimation:

$$A_t = \sigma_t + (\gamma\lambda)\sigma_{t+1} + (\gamma\lambda)^2\sigma_{t+2} + ... + (\gamma\lambda)^{T-t+1}\sigma_{t-1}$$

where $\sigma_t = r_t + \gamma V(s_{t+1}) - V(s_t)$. It's possible to notice that the original A2C estimation is a special case of the above one with $\lambda = 1$. Concluding, PPO uses also a long sequence of samples obtained from the environment and then estimate the advantage for the whole sequence before the performance of several epochs of training.

In our implementation the PPO method didn't result in an effective approach, as the policy continued to converge into a situation where one single action had almost all the probability of being chosen. To tackle this situation we tried to increment the entropy weight in the loss function, which gave some better results in the first steps of training but without improving the situation in a meaningful way.

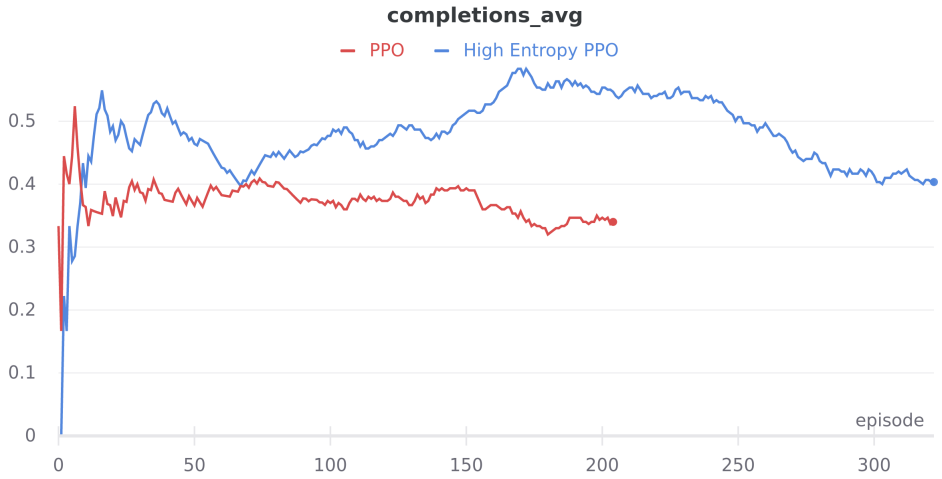

Figure 4: PPO Training - Comparison between different entropy weights

2.3 Deep Q-Networks (DQN)

DQN [3] are the tool used for applying Deep Q-Learning. The main idea is to unify Q-Learning and Deep Learning for dealing with complex environments. To achieve that, a neural network is used as a predictor of the $Q(s, a)$ function.

To achieve an exploration behaviour a ϵ parameter with a decaying factor is used as a probability of performing random actions. The more time passes (resulting in the network being trained more) the lower the probability of exploring. This technique is called *exploration vs exploitation*.

By performing training on subsequent states the dataset wouldn't be ideally distributed and samples are not independent. To deal with that issue, training is done over a large buffer of the past experiences used to sample training data from it. This technique is called **experience replay**. The simplest implementation of it keeps a fixed size, having new data added to the end of the buffer for every agent step, such that it pushes the oldest experience out of it.

The DQN algorithm consists of the following main steps:

1. Initialize the parameters for $Q(s,a)$ and $\hat{Q}(s,a)$ with random weights, $\epsilon \leftarrow 1.0$, and empty the replay buffer.
2. With probability ϵ, select a random action, a, otherwise, $a = argmax_a Q(s,a)$.
3. Execute action a in an emulator and observe the reward, r, and the next state, *s'*.
4. Store transition *(s,a,r,s')* in the replay buffer.
5. Sample a random mini-batch of transitions from the replay buffer.
6. For every transition in the buffer, calculate target $y = r$ if the episode has ended at this step, or $y = r + \gamma max_{a' \in A} \hat{Q}(s', a')$ otherwise.
7. Calculate loss: $\mathcal{L} = (Q(s,a) - y)^2$
8. Update $Q(s,a)$ using the SGD algorithm by minimizing the loss with respect to the model parameters.
9. Every N steps, copy weights from Q to $\hat{Q}$.
10. Repeat from step 2 until converged.

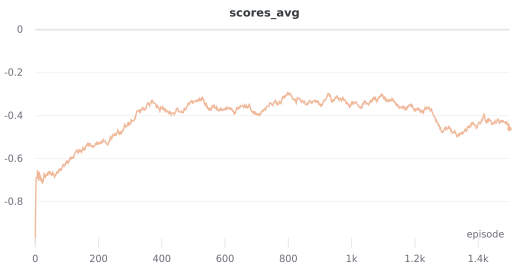

Fig. 5: *DQN Training - Score*

Fig. 6: *DQN Training - Completions*

2.3.1 Double DQN

Double DQN [4] is a simple tweak to the basic method which consists in implementing a second neural network (called *target network*) with the same architecture of the main one, which is used to predict the Q-values for the training step, while the back-propagation is performed on the main network. This technique allows to reduce the Q-values overestimation problem which afflicts vanilla DQN.

The target network is updated every once in a while and the number of steps between the updates is a tunable hyperparameter.

$$Q(s_t, a_t) = r_t + \gamma max_a Q'(s_{t+1}, argmax_a Q(s_{t+1}, a))$$

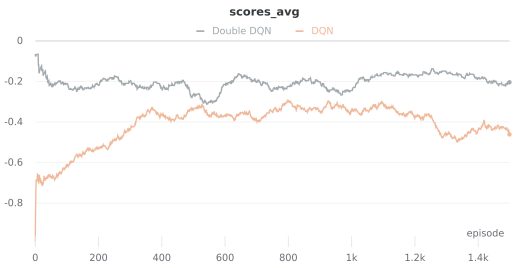

Fig. 7: *Double-DQN (H.U.) - Score*

Fig. 8: *Double-DQN (H.U.) - Completions*

An interesting variation concerns the usage of a soft update instead of a periodic hard update, which in our case improves slightly the effectiveness of the agents.

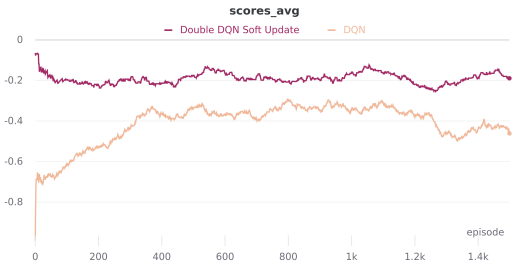

Fig. 9: *Double-DQN (S.U.) - Score*

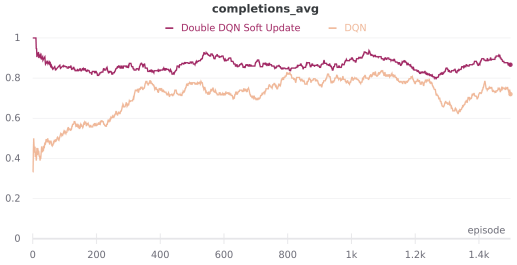

Fig. 10: *Double-DQN (S.U.) - Completions*

2.3.2 Dueling DQN

Q-values, $Q(s, a)$, that our network is trying to approximate can be divided into quantities: the value of the state, $V(s)$, and the advantage of actions in this state, $A(s, a)$. The advantage is supposed to bridge the gap from $A(s)$ to $Q(s, a)$, by definition, $Q(s, a) = V(s) + A(s, a)$. In other words, the advantage, $A(s, a)$ is just a delta, saying

how much extra reward some particular action from the state brings us. The dueling DQN explicitly separate the value of the state and the advantage into different network's architecture, which brought better training stability and faster convergence. The classic DQN network takes features from the layer and, using dense layers, transforms them into a vector of Q-values, one for each action. Instead, dueling DQN takes features and process them using two independent paths, one path responsible for $V(s)$ prediction (a single number) and the other path predicts individual advantage values, having the same dimension as Q-values in the classic case [5]. After that, $V(s)$ is added to every value of $A(s, a)$ to obtain $Q(s, a)$. This constraint could be enforced in different ways, via the loss function or directly by affecting the Q expression subtracting the mean value of the advantage.

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{N}\sum_{k} A(s,k)$$

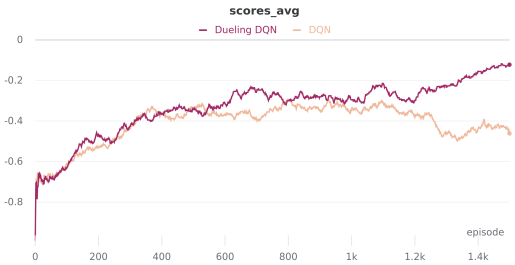

Fig. 11: *Dueling DQN Training - Scores*

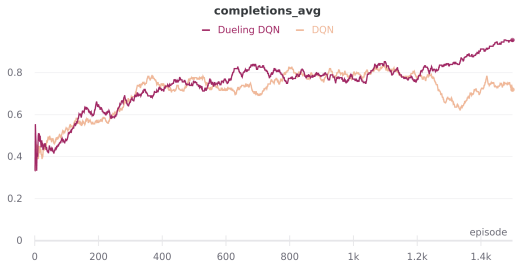

Fig. 12: *Dueling DQN Training - Completions*

2.3.3 Noisy Networks

We've talked about the exploration environment via the *epsilon-greedy method*. This process works well for simple environments with short episodes. In Noisy Network's paper [6] the author proposed a simple but well working solution: add noise to the weights of fully connected layers and adjust the parameters of this noise during training using backpropagation.

There are two ways of adding the noise, both of them work according to their experiments, but they have different computational overheads:

1. Independent Gaussian noise: for every weight in a fully connected layer, we have a random value that we draw from the normal distribution. Parameters of the noise, μ and σ, are stored inside the layer and get trained using backpropagation in the same way that we train weights of the standard linear layer. The output of such a "noisy layer" is calculated in the same way as in a linear layer.

2. Factorized Gaussian noise: to minimize the number of random values to be sampled, the authors proposed keeping only two random vectors: one with the size

of the input and another with the size of the output of the layer. Then, a random matrix for the layer is created by calculating the outer product of the vectors.

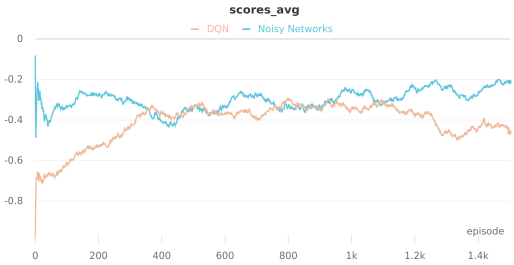

Fig. 13: *Noisy Network Training - Scores*

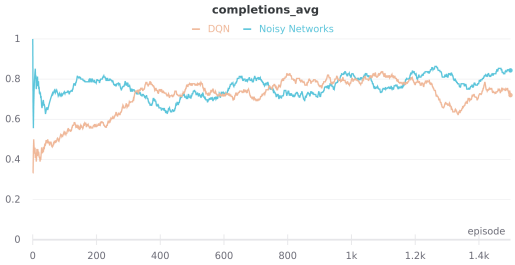

Fig. 14: *Noisy Network Training - Completions*

2.3.4 Prioritized Experience Replay

The classic DQN can be improved in all aspects, also, in how to deal with the experience replay. Prioritized replay buffer [7] works by assigning priorities to buffer samples, according to training loss and sampling from it proportionally to the priorities. The basic idea of PER is to train on data that "surprised" the system. The sensitive point, is to maintain balance between training on "unusual" samples and training on the rest of the buffer. The priority of every sample in the buffer is:

$$P(i) = \frac{p_i^\alpha}{\sum_k P_k^\alpha}$$

Where p_i is the priority of the *i-th* sample in the buffer and α is a parameter that expresses the importance we give to the priority. If $\alpha = 0$, our sampling will become uniform as in the classic DQN method. Larger values of α put more stress on samples with higher priority. It is general usage to start with $\alpha = 0.6$. By adjusting the priorities for the samples, bias is introduced into our data distribution (where we sample some transition much more frequently than others) that need to be compensated for SGD to work. To get this result, the priority value is modified as follows:

$$w_i = (N \cdot P(i))^\beta, \quad \beta \in [0, 1]$$

If $\beta = 1$, the bias introduced by the sampling is fully compensated. Anyway, the general usage it to start with a random value between 0 and 1 and slowly increase it to 1 during the training.

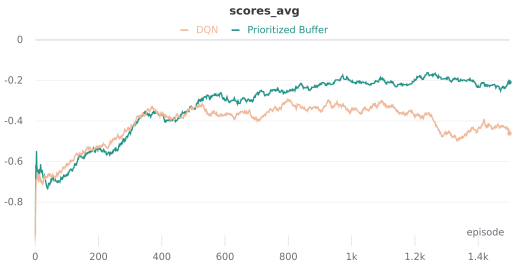

Fig. 15: *Prioritized Experience Buffer Training - Scores*

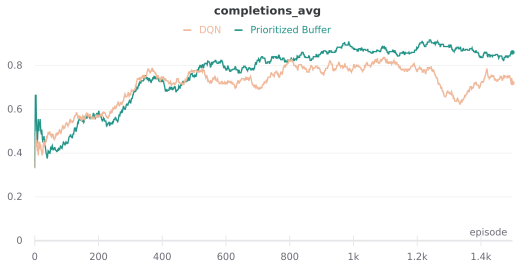

Fig. 16: *Prioritized Experience Buffer Training - Completions*

3 Results

3.1 Test Environments

The proposed solutions have been tested with different environment setups, mainly divided in N levels with growing complexity:

- L1:

Parameter	Value
n_agents	5
x_dim	20
y_dim	30
n_cities	2
max_rails_between_cities	2
max_rails_in_city	3
min_malfunction_interval	50
grid_mode	False
malfunction_duration	[20, 50]

- L2:

Parameter	Value
n_agents	7
x_dim	30
y_dim	40
n_cities	5
max_rails_between_cities	2
max_rails_in_city	3
min_malfunction_interval	50
grid_mode	False
malfunction_duration	[20, 50]

- L3:

Parameter	Value
n_agents	10
x_dim	40
y_dim	60
n_cities	9
max_rails_between_cities	4
max_rails_in_city	3
min_malfunction_interval	50
grid_mode	False
malfunction_duration	[20, 50]

3.2 Hyper-Parameters Tuning

The last test consists in performing hyperparameter tuning in order to find the most promising set of parameters for the final training.

We achieved that by performing two different tuning sessions through WandB: the first one by iterating randomly through a large parameter space and the second by performing a grid search over all the given parameters. The results are the following.

Parameter	Values
Discount Factor (γ)	0.99, 0.9
Learning Rate (α)	0.0001, 0.0002, 0.0005
Noisy Network	True, False
Soft Update Rate (τ)	0.05, 0.075, 0.1, 0.2

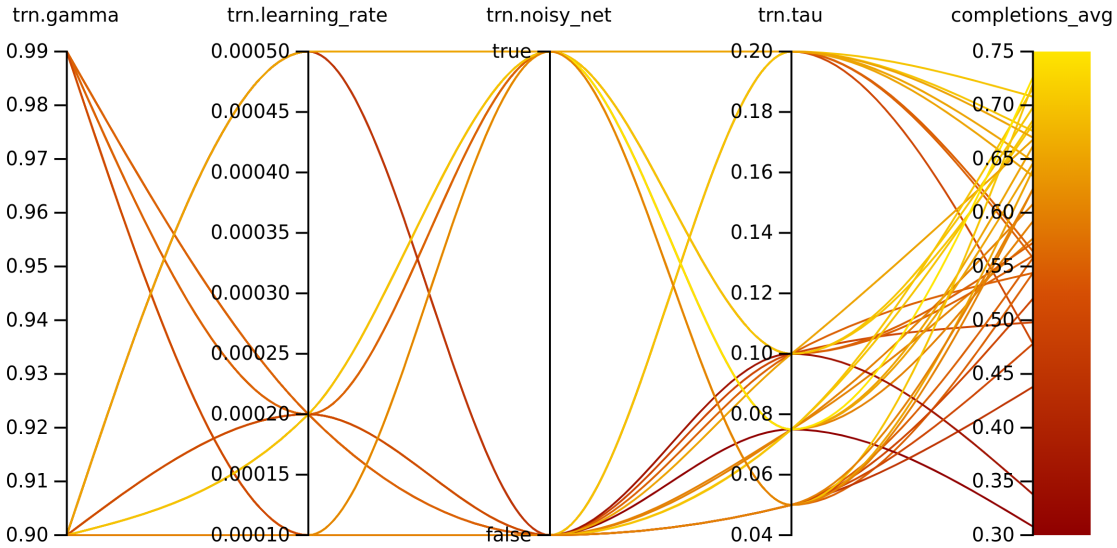

Figure 17: Random Search Sweep

Parameter	Values
Tree Observation Depth	2, 4, 6
Batch Size	32, 64
Learning Rate	0.0001, 0.0005
Noisy Network	True, False

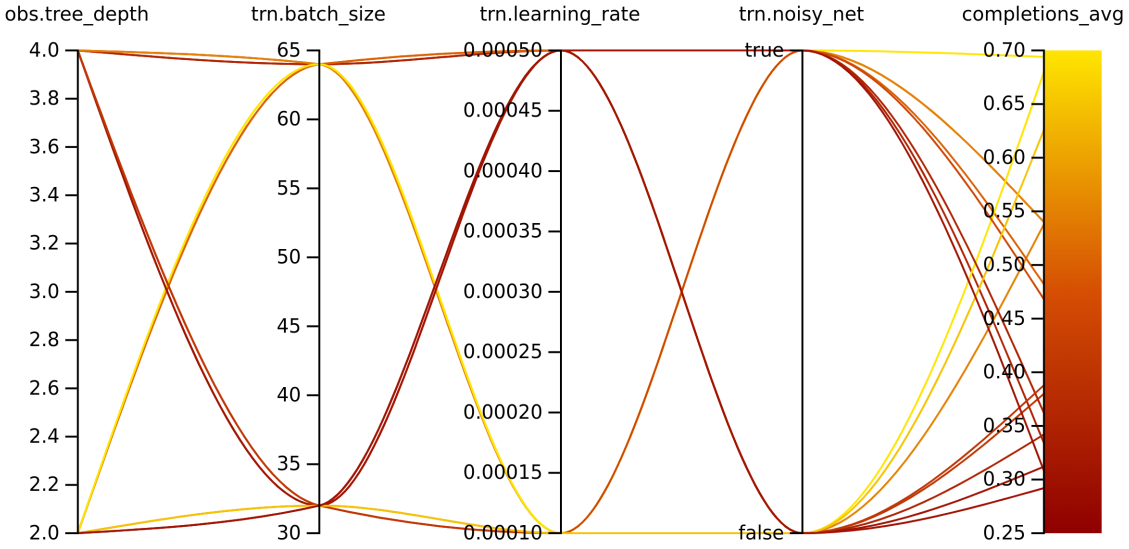

Figure 18: Grid Search Sweep

By considering the completions percentage we could infer the optimal parameters for the DDDQN architecture.

Parameter	Values
Tree Observation Depth	2
Batch Size	64
Learning Rate	0.0001
Noisy Network	True
Discount Factor (γ)	0.9
Soft Update Rate (τ)	0.2
ϵ Decay	0.997

3.2.1 Performances

Figure 19: Performances on L1

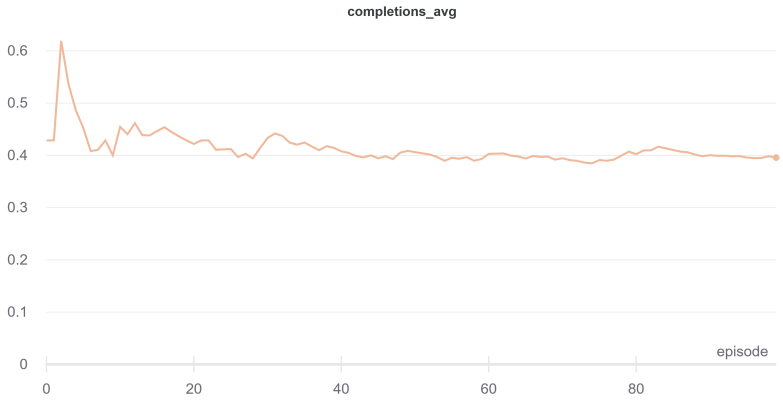

Figure 20: Performances on L2

Figure 21: Performances on L3

4 Conclusions

After studying the problem, we planned to start with an Actor-Critic approach implementing from scratch a PPO network. Having a baseline for A-C approach, we started developing a DQN to state which of the two classic approach was the better performing. We performed several tests and logged on *WandB* [8] platform the results. It ended up that DQN approach turned out having better outcomes. Therefore, we decided to introduce major improvements for our DQN approach, such as: Double-DQN and Dueling-DQN. In the end we unified those development obtaining our DDDQN which performed better then the previous ones. Thus, we implemented further tweaks on DDDQN, implementing a Prioritized Experience Buffer and implementing a general use case *predictor*.

4.1 Future Development

In future development it would be interesting to consider improving the PPO:

- with a *Truly Proximal Policy Optimization* [9], which adopts a new clipping function to support a rollback behaviour to restrict the difference between the new policy and the old one and it replaces the triggering condition for clipping with a trust region-based one. This approach should guarantee monotonic improvement;
- another possible improvement could be a *Trust Region-Guided Proximal Policy Optimization* [10] which aims to avoid the risk of lack of exploration by adaptively adjust the clipping range within the trust region. In this way the PPO should improve the exploration ability and enjoy better performance with respect to the original PPO.

Moreover, regarding memory management, a possible improvement could be the *Hindsight Experience Replay* [11] which allows sample-efficient learning from rewards which are sparse and binary, avoiding the need for complicated reward engineering which could be combined with an arbitrary off-policy method.

References

[1] *Flatland Challenge 2021*. URL: https://www.aicrowd.com/challenges/flatland.

[2] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[3] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[4] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: 1509.06461 [cs.LG].

[5] Ziyu Wang et al. *Dueling Network Architectures for Deep Reinforcement Learning*. 2016. arXiv: 1511.06581 [cs.LG].

[6] Meire Fortunato et al. *Noisy Networks for Exploration*. 2019. arXiv: 1706.10295 [cs.LG].

[7] Tom Schaul et al. *Prioritized Experience Replay*. 2016. arXiv: 1511.05952 [cs.LG].

[8] *Weights & Biases*. URL: https://wandb.ai/.

[9] Yuhui Wang et al. *Truly Proximal Policy Optimization*. 2020. arXiv: 1903.07940 [cs.LG].

[10] Yuhui Wang et al. *Trust Region-Guided Proximal Policy Optimization*. 2019. arXiv: 1901.10314 [cs.LG].

[11] Marcin Andrychowicz et al. "Hindsight Experience Replay". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/453fadbd8a1a3af50a9df4df899537b5-Paper.pdf.