Alma Mater Studiorum · University of Bologna

Flatland Challenge
Deep learning course final project

Lorenzo borelli (lorenzo.borelli@studio.unibo.it)
Giovanni Minelli (giovanni.minelli2@studio.unibo.it)
Lorenzo Turrini(lorenzo.turrini4@studio.unibo.it)

August 24, 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The **Flatland challenge** consists in tackling the vehicle scheduling problem, in a controlled and simulated environment. The main approach used here is reinforcement learning, however there are several other possibilities that fall under the umbrella of combinatorial optimization.
The aim of the challenge, from a RL standpoint, is to allow the agents to learn how to optimally reach their target in minimal time.

Chapter 2

The environment

2.1 Railway network

The flatland environment is simulated by a rectangular grid of fixed size, which can be set by the user. Each cell of the grid is either a rail cell, or an empty unusable cell, or a **target** cell. Rails are of different type, depending on **transitions**: there are 16 different transitions in flatland, since there are 4 different orientations of the agent, and 4 other directions of exit from the cell. Thus, each cell is equipped with a bitmap that represents the whole transition space.
However, not every transition is allowed in flatland, since the aim is to actually simulate a real railway system: effectively, only 2 exit directions are allowed from every orientation of the agent, which result in 8 different cell types (including empty ones).

Flatland offers also a *sparse_rail_generator* that randomizes the creation of a realistic railway structure, where clusters of cities are sparsely connected to each other, allowing to mimic as faithfully as possible real city railway networks.

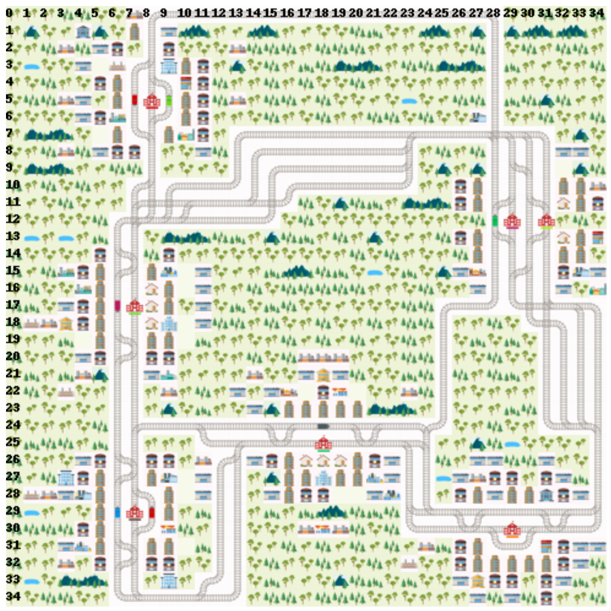


Figure 2.1: Sparse railway

2.2 Agents

Trains have a number of important properties:

- **Position**: the current coordinates of the agent.
- **Target**: the position of the target cell.
- **Direction**: the current orientation, with 0 corresponding to North, 1 to East, 2 to South, 3 to West.
- **Movement**: a flag that tells whether the agent is moving or not.

Since every agent is liable to malfunctions, much like real trains, there are properties to keep track of that, as well as variables that store the agents' speed:

- **Malfunction rate**: the Poisson rate at which malfunctions occur
- **Malfunction**: a counter of the remaining time the agent will remain malfunctioning

- **Next malfunction**: number of steps until next malfunction
- **Number of malfunctions**: the total number of malfunctions for this agent
- **Max speed**: a fraction between 0 and 1: a speed of 1/2 means that the agent changes cell after 2 time steps.
- **Position fraction**: related to speed, indicates when the next action can be taken

2.3 Environment Actions

The available actions are:

- **DO NOTHING (0)**: Default action if None has been provided or the value is not within this list. If agent.moving is True then the agent will MOVE FORWARD.
- **MOVE LEFT (1)**: If agent.moving is False then becomes True. If it's possible turn the agent left, changing its direction, otherwise if agent.moving is True tries the action MOVE FORWARD.
- **MOVE FORWARD (2)**: If agent.moving is False then becomes True. It updates the direction of the agent and if the new cell is a dead-end the new direction is the opposite of the current.
- **MOVE RIGHT (3)**: If agent.moving is False then becomes True. If it's possible turn the agent right, changing its direction, otherwise if agent.moving is True tries the action MOVE FORWARD.
- **STOP MOVING (4)**: If agent.moving is True then becomes False. Stop the agent in the current occupied cell.

2.4 Rewards

The rewards are based on the following values:

- invalid action penalty which is currently set to 0, penalty for requesting an invalid action
- **step penalty** which is -1 * alpha, penalty for a time step.
- **global reward** which is 1 * beta, a sort of default penalty.
- stop penalty which is currently set to 0, penalty for stopping a moving agent
- start penalty which is currently set to 0, penalty for starting a stopped agent

The full step penalty is computed as the product between step penalty and agent.speed data['speed']. There are different rewards for different situations:

- single agents that are in DONE or in DONE REMOVED have zero reward.
- all agents that have finished in this episode (checked at the end of the step) or previously (checked at the beginning of the step), have reward equal to the global reward (when in step all agents have reached their target)
- full step penalty is assigned when an agent is READY TO DEPART and in the current turn moves or stay there (2th step agent case), or when is in malfunction.
- full step penalty plus the other penalties (invalid action penalty, stop penalty and start penalty) when the agent is finishing actions or start new ones. Currently the other penalties are all set to zero.

The end of the Each train starts counting rewards since the beginning, not since it becomes ACTIVE. Currently it is possible to say that agents' rewards are always full step, excluding when the episode ends and when they have finished, where is 0.

Chapter 3

The our environment

Our flatland environment is based on RailEnv where we wrapped the main functionalities, methods and we added parameters in order to personalizehave parameterized it through file yaml *"env_parameters.yml"*. We have wrapped and added these main functionalities:

- **reset:**
- **step:**
- **render env:**
- **encode info:**

3.1 Parameters

3.2 Metrics

3.3 Evaluation and platforms

3.4 Action

The Flatland environment provides for each agent five different actions that they are described in 2.3.

The DO NOTHING is not necessary to reach a solution, because the agent can continue moving forward deciding each time the action MOVE FORWARD or stop using STOP MOVING. As we think that further from being useless it may also damage the overall performance we decided to consider its removal. It has already mentioned the ambiguity of the actions MOVE LEFT and MOVE RIGHT where they are forbidden, it is natural to conclude that the agents may learn bad policies that maps these actions to the same effect of the MOVE FORWARD action. We observed that this phenomenon is very common due to the presence of long straight paths where the agent is allowed only to stop or move forward and concluded that even stopping in the middle of the rail does not have much sense because agents still stop when other agents block their way due to deadlocks, different speeds or malfunctions.
For this reason we considered the possibility to force agents to only decide and learn in switches, where multiple actions are allowed and agents may learn to give way to other agents, avoid deadlocks, reach the target and more. This considerations lead to skip a lot of choices during learning and deploy Action Masking to avoid illegal actions in fact some studies have proposed to deploy action masking to avoid the selection of multiple actions when they are not necessary. A very common strategy to address invalid actions is applying negative rewards, but this also requires the agent to explore the actions and understand how to map actions to the possibility of applying them. During this period it is possible that the agent converges to a wrong policy. Invalid action masking helps to avoid sampling invalid actions by "masking out" the network outcomes corresponding to the invalid actions.
Our approach is been to evaluate the action only before or on the switch and in these case there is a flag "decision_required" equals true, otherwise all other case this flag is false.

- **"decision_required"** = **True**: In this case, action is decided by policy on based the observation.
- **"decision_required"** = **False**: In this case, action is decided by environment and it is constant "RailEnvActions.MOVE_FORWARD"

so the agent will continue on the road that it has begun to run across in the same direction.

3.5 Deadlock Controller

3.6 Rewards

Chapter 4

Reinforcement learning

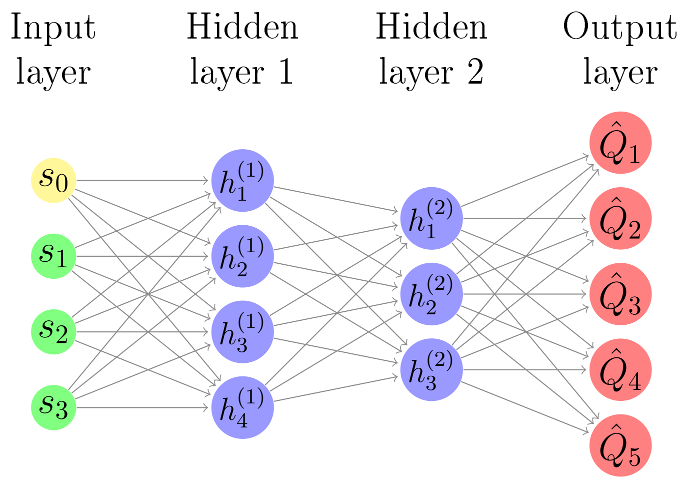
Reinforcement learning aims at training agents that operate in an environment by assigning rewards to their actions. Agents decide which action to take based on a **policy** $\pi(a_t, s_t)$, with s_t being the observation of the environment at time step t.
Maximizing the sum of rewards is the method whereby the agent improves the policy and learns.

4.1 DQN

[1] detailed a network model to approximate the Q-function $Q^{\pi}(s, a)$, which measures, for each state-action pair, the discounted sum of rewards, following from the policy π. The optimal Q-function, that is the maximum reward which can be obtained by performing a certain action a in state s, obeys the **Bellman optimality equation**

$$Q^{*}(s, a) = E[r + \gamma max_{a'} Q^{*}(s', a')]$$

It follows that the maximum return is obtained by the immediate reward and the discounted return that the agent gets by following the policy; in practice, however, neural networks are needed in order to avoid computing a huge Q-table, and they are trained by minimizing the **temporal difference** loss. We started by implementing a slight variation of DQN as described in [1], without the use of CNN, but by simply using fully connected layers with **ReLU**, since we felt that this specific environment wouldn't benefit from classic image convolution, though we used graph convolutions later.



The units in the hidden layers default to 24 and 12, but the dimensions are fully customizable by the user, while the input layer is just made by the state of the environment, and the output layer has as many layers as there are actions in flatland. The model was also enriched with a few improvements:

- **Experience replay**: past experiences are recorded in a memory buffer that is sampled, when needed, to train the network. We implemented two different variations: random and **prioritized experience replay**, as explained in [5]. The paper illustrates greedy and stochastic prioritization: both leverage priority based on **temporal difference**, but the latter is more robust, allowing to replay every experience in the buffer at least once. We implemented **proportional stochastic prioritization**, which uses a small positive hyperparameter ϵ to always keep a non-zero priority, thus allowing to revisit even the lowest priority experiences. Concretely, the probability to sample transition i is

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

  with $p_i = temporal\ difference\ + \epsilon$. This implementation uses a *sum tree* data structure to store priorities and experiences, a binary tree where the leaves contain the priorities and data, while the inner nodes feature the sum of the children's priorities, with the root having the total priority: the sum tree provides $O(log(n))$ updates and sampling, with an efficient way to retrieve cumulative sums. Finally, for further stability, we added **importance-sampling weights**: for each transition i, weights are computed as

$$w_i = (\frac{1}{N} * \frac{1}{P(i)})^\beta$$

, and then fed to the Q-learning update, giving higher importance to low priority transitions, thus fixing the inherent bias of priority sampling. β is used to control the amount of sampling over time, and usually reaches 1 by the end, since it's more important to have unbiased updates near convergence.

- **Fixed Q-targets**: since in a reinforcement learning setting we do not have target values prepared, we would need to use the same network for both the model's weights and for the target, leading to instability of the target values. Thus, we use 2 equal networks, one to just compute the target Q value, and the other for training. Every 100 time step, a tunable hyperparameter, the target network is *soft updated* with a portion of the current weights determined by this formula

  $$\theta_i = \tau * \theta_i + (1 - \tau) * \theta_i$$

  and by the hyperparameter τ. Furthermore, the model itself is updated every 4 time steps, which are customizable as well, so as to increase training speed.

- **Huber loss**: as explained in [2], this loss function clips the gradient to the $[-1, 1]$ interval, avoiding **exploding gradient**. While on one hand it behaves like the absolute loss, for small errors it is more similar to the squared loss, thus combining the properties of both losses, while circumventing the problem of outliers which plagues the squared loss.

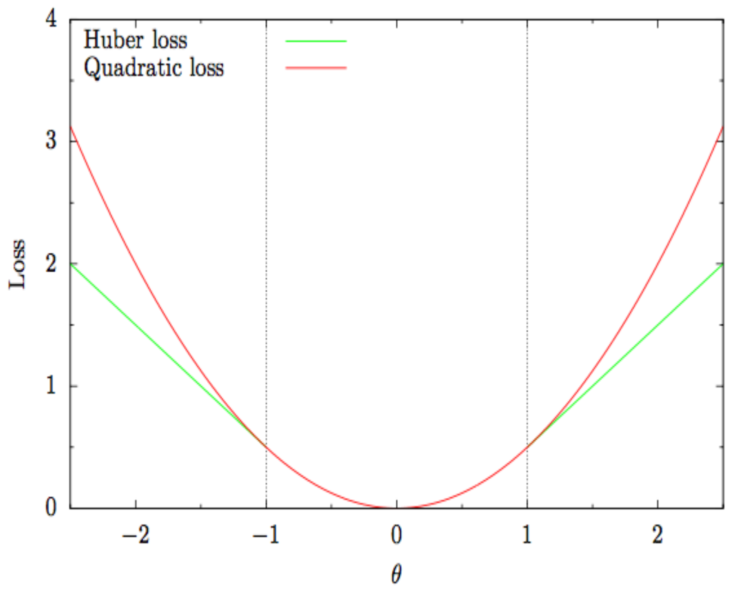
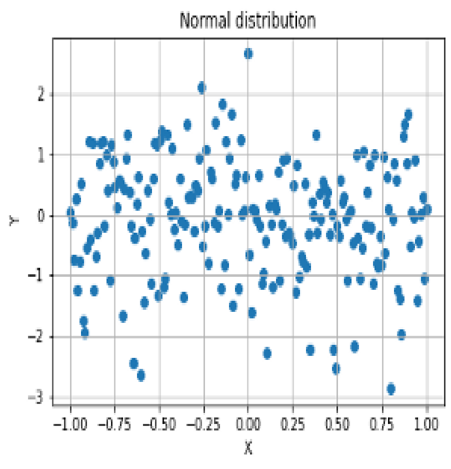


Figure 4.1: Huber and squared loss

- **Kaiming initialization**: a method for weights matrix initialization, described in [3], which draws samples from a standard normal distribution, to avoid **vanishing** and **exploding gradient**, and then adjusts this distribution to the ReLU activation function, by doubling the variance, since ReLU halves it w.r.t. to the original standard normal distribution (see 4.2).



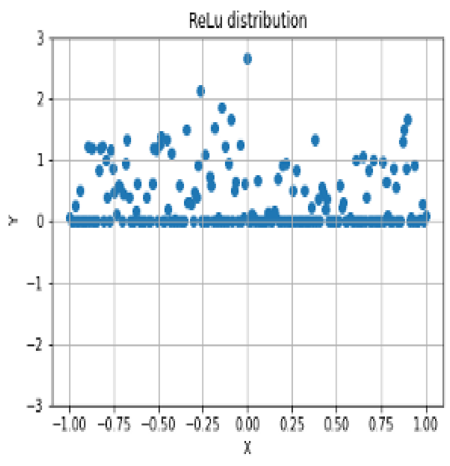


Figure 4.2: Normal and ReLU distributions

4.1.1 Dueling

The model was further improved with **dueling** architecture. The main issue of standard DQN lies in the fact that the **Bellman operator** tends to overestimate the Q value: as stated in [4], the solution seems to be to separate the computation of the Q value in 2 separate functions,

$$Q_\pi(s,a) = V_\pi(s) + A_\pi(s,a)$$

The **advantage** $A(s,a)$, which computes the ”advantage” of taking the action a in state s w.r.t. the other possible actions in that state, and the **value** $V(s)$ which simply measures how good the state s is, regardless of the possible actions; this is particularly useful whenever we're dealing with environments that are not affected by actions in every state, like Flatland, where actions are only really relevant at switches.

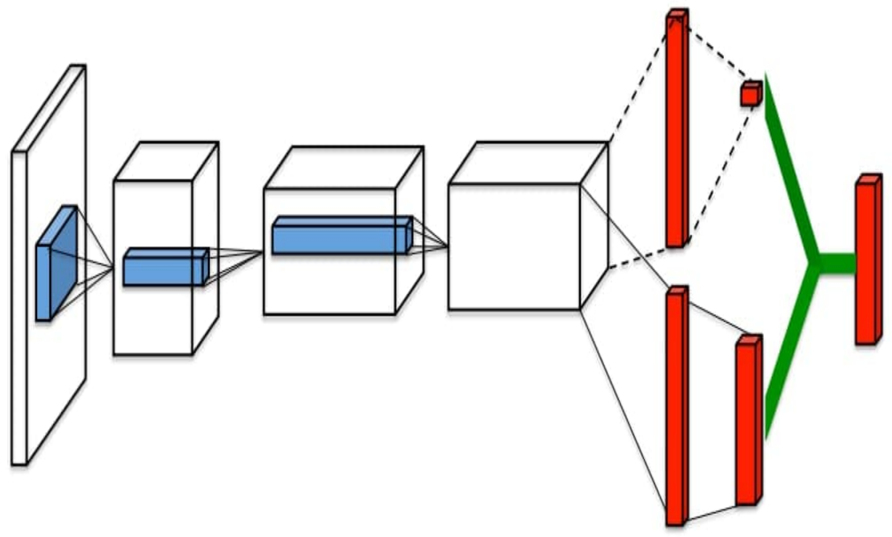


Figure 4.3: Dueling DQN

Advantage and value still need to be recombined, and since the standard formula doesn't allow to extract A and V from the Q value, [4] suggested to force the advantage to be 0 at the chosen best action, which results in the output formula for Q

$$Q_\pi(s,a) = V_\pi(s) + (A_\pi(s,a) - max_{a' in A} A(s,a'))$$

It also highlights, however, that empirically, subtracting the average advantage yields better results.

Chapter 5

Observation

5.1 GNN

Given the use of a graph observation, we needed a mechanism to normalize it in order to then feed it to the reinforcement learning algorithm. GNN are typically used to create an embedded representation of node features, which is then used for another task, like classification or regression.

Specifically, we wanted to fully take advantage of the connections between the nodes, so that every node's representation, after being processed by the GNN, would in fact include data about the neighborhood. Thus, we employed **Graph Convolutional Networks**, as described in [6]: GCN perform a convolution on graph features, by using an update rule

$$H^{(l+1)} = \sigma(D^{\frac{-1}{2}} A' D^{\frac{-1}{2}} H^l W)$$

that, at each layer, effectively combines together features of neighboring nodes.
H^0 is the input feature matrix that contains features for each node, W is a layer-specific weight matrix, while the **adjacency matrix** A is actually summed with the idendity matrix in order to obtain A', which considers also self-connections in the graph, otherwise each node in the

GCN's output would never include information about themselves.
Furthermore, the **degree matrix** D is included for *symmetric normalization*, since the matrix multiplication risks scaling up the features.
Everything is then passed to an non-linear activation function σ, and repeated for a number of convolution layers which is a specific parameter of the model.

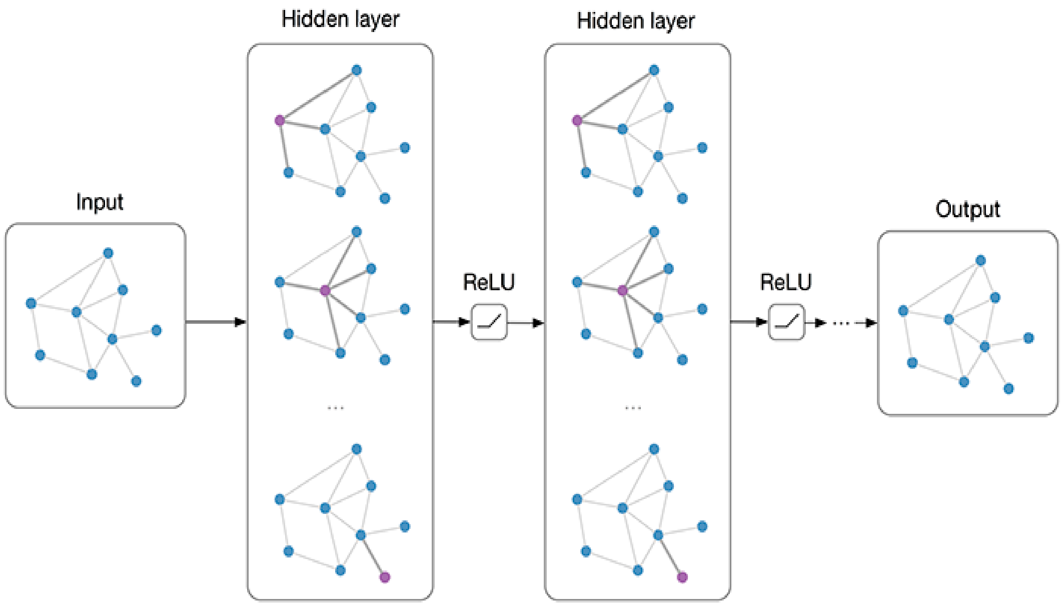


Figure 5.1: GCN

As features, each node has the total **distance** from every adjacent switch in the graph observation, and a one-hot encoding of its type (starting or target node, conflict, deadlock, starvation node, or other).
After the input node features have been processed by the gcn, agent-specific features are added to the vector representation that is then passed to the dqn.

Bibliography

[1] Mnih et Al. *Playing Atari with Deep Reinforcement learning.* In: (2013). cite arxiv:1312.5602 NIPS Deep Learning Workshop 2013. url: `http://arxiv.org/abs/1312.5602`.

[2] Mnih et Al. *Human Level Control Through Deep Reinforcement Learning.* Nature, 518, pages 529–533, 2015.

[3] He et Al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.* ICCV '15: Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV), 2015. pp. 1026–1034 `https://doi.org/10.1109/ICCV.2015.123`

[4] Wang, Ziyu and Schaul, Tom and Hessel, Matteo and Hasselt, Hado and Lanctot, Marc and Freitas, Nando *Dueling Network Architectures for Deep Reinforcement Learning* Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48. ICML'16.New York, NY, USA: JMLR.org, 2016, pp. 1995–2003.

[5] T. Schaul, J. Quan, I. Antonoglou, and D. Silver *Prioritized Experience Replay* In: (2016) cite arxiv:1511.05952 Comment: Published at ICLR 2016.

[6] Thomas N. Kipf, Max Welling *Semi-Supervised Classification with Graph Convolutional Networks* ICLR 2017. cite arXiv:1609.02907