Alma Mater Studiorum · University of Bologna

Flatland Challenge

Deep learning course final project

Leonardo Calbi (leonardo.calbi@studio.unibo.it)
Alessio Falai (alessio.falai@studio.unibo.it)

December 21, 2020

Contents

List of Figures

Foreword

The Flatland challenge is a competition organized by AIcrowd [1] with the help of SBB (Swiss Federal Railways) to foster innovation with what regards the scheduling of trains trajectories in a railway environment.

As reported on the official challenge website, SBB operates the densest mixed railway traffic in the world. It maintains and operates the biggest railway infrastructure in Switzerland: today, as of 2020, there are more than 10 000 trains running each day, being routed over 13 000 switches and controlled by more than 32 000 signals.

The Flatland challenge aims to address the vehicle rescheduling problem by providing a simplistic grid world environment and allowing for diverse solution approaches. In particular, the first edition of the challenge was hosted during 2019 and the submitted solutions were mainly based on OR (Operation Research) methodologies, while the second edition of the competition, i.e. the NeurIPS 2020 edition, had the goal of favoring the implementation of RL (Reinforcement Learning) based solutions.

1 Introduction

At the core of this challenge lies the general vehicle rescheduling problem (VRSP) proposed by Li, Mirchandani, and Borenstein in 2007 [6]:

> *The vehicle rescheduling problem (VRSP) arises when a previously assigned trip is disrupted. A traffic accident, a medical emergency, or a breakdown of a vehicle are examples of possible disruptions that demand the rescheduling of vehicle trips. The VRSP can be approached as a dynamic version of the classical vehicle scheduling problem (VSP) where assignments are generated dynamically.*

The problem is formulated as a 2D grid environment with restricted transitions between neighboring cells to represent railway networks. On the 2D grid, multiple agents with different objectives must collaborate to maximize the global reward.

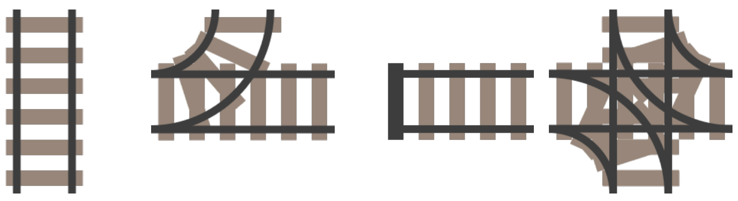
The overall goal is to make all agents (trains) arrive at their target destination with a minimal travel time. In other words, we want to minimize the time steps (or wait time) that it takes for each agent in the group to reach its destination.

2 Background

Railway

As already pointed out, the Flatland environment is represented as a 2D grid of dimension $W \times H$ and each cell in the grid can be one of many different types. The different types of cells can belong to the following categories: `rail` and `empty`.

The `rail` cells are the most intricated of the two, in that there exists different types of them. In particular, figure 2.1 shows examples of possible `rail` cells that can be used to build up a railway environment in Flatland. Other than the ones shown in figure 2.1 there are also diamond crossings (i.e. two orthogonal straight rails crossing each other), single slip switches (i.e. the same as double slip switches but with a single choice) and symmetrical switches (which are special kinds of switches that bifurcate to a left and right branch). Moreover, every `rail` cell can be rotated by 90° and mirrored along both axis, to allow more combinations between them to be made, in order to guarantee a greater degree of diversity between different railways.



(a) Straight (b) Simple switch (c) Deadend (d) Double slip switch

Figure 2.1: Different `rail` cell types

Moreover, `rail` cells can be occupied by the following entities (the ones shown in figure 2.2):

- Agent: one `rail` cell can be seen as a resource with availability equal to one, so that in each time step only one agent can occupy it
- Target: each target is statically assigned to one `rail` cell. Target cells represent the destination of one or more agents (different agents could have the same target). Moreover, the number of possible targets present in the environment is clearly limited by the number of agents

An important fact about the different types of `rail` cells is that only switches require an agent to make a choice. In Flatland (like in reality) a maximum of two options is available. There does not exist a switch with three or more options.

Finally, every cell that is not `rail` is `empty` and neither targets nor agents can

(a) Agent (b) Target

Figure 2.2: Agents and targets

fill it up. As shown in figure 2.3, it is interesting to notice that Flatland is a very sparse environment, meaning that there are a lot more `empty` cells than `rail` ones: because of this, representing the environment as a simple dense matrix could lead to overheads and efficiency issues, especially when dealing with relatively big environments.

In the end, the only cell types that we care about are the `rail` ones: the `empty` ones are useful only for visualization purposes. Because of this, we could think of representing the environment as a sparse matrix or as a graph containing only `rail` cells or a subset of them (we will address this issue in section 3.1).

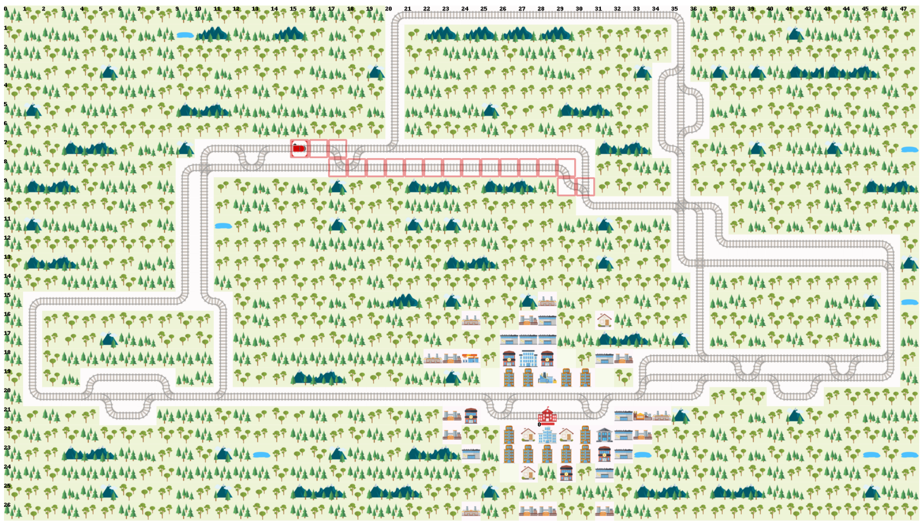


Figure 2.3: An example of a railway environment

Transitions

An agent in the Flatland environment is a train that starts from a random `rail` cell in the map and has to arrive to its assigned target in the minimum number of steps. To do so, the agent can only occupy `rail` cells.

To move from a cell to another one the agent has to make a choice and, depending on the cell type that they are on and on the connections between cells, an agent can transition from cell i, when looking towards direction d_i, to cell j, looking towards direction d_j, if and only if $T_i(d_i, d_j) = 1$, where T_i is the transition matrix associated to cell i, s.t. $T_i(d_i, d_j) = 0$ means that the transition from cell i, direction d_i to cell j, direction d_j is forbidden (likewise $T_i(d_i, d_j) = 1$ means that the transition is allowed). Directions d_* are represented as the 4 cardinal directions, i.e. North, East, South and West (N, E, S, W), so that each transition matrix T_* can be characterized as a 4×4 binary matrix. For example, a deadend cell like the one reported in figure 2.1c would have a transition matrix like the one shown in figure 2.4.

$$\begin{array}{c c} & \begin{array}{cccc} N & E & S & W \end{array} \\ \begin{array}{c} N \\ E \\ S \\ W \end{array} & \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \xrightarrow[\text{bitmap}]{\text{to}} \text{0000 000}\mathbf{1}\text{ 0000 0000}$$

Figure 2.4: Transition matrix and bitmap of a deadend

As we can observe, only one entry in the matrix has value 1, meaning that only one transition is possible, i.e. the one s.t. the agent enters heading East and exits heading West.

In the Flatland library, a transition matrix is represented by a bitmap, which can be seen as a linearization by rows of the reported matrix (the mapping between the transition matrix of the deadend cell 2.1c and its bitmap is again shown in figure 2.4). In this way, by simply counting the number of true values in the bitmap, we can understand the type of `rail` cell that we are examining (e.g. only one true value indicates a deadend, while exactly two true values indicate a straight rail).

Actions

Flatland has a discrete action space, meaning that only 5 possibilities have to be considered at each transition. In particular, Flatland uses the following convention:

1. `MOVE_FORWARD`: the agent maintains the current movement direction, if possible (i.e. if it was heading North, it will continue heading North)
2. `MOVE_LEFT`: if the agent is at a switch with a transition to its left, the agent will choose the left path, otherwise the action has no effect (e.g. if the agent was heading North, it will be directed towards East)
3. `MOVE_RIGHT`: if the agent is at a switch with a transition to its right, the agent will choose the right path, otherwise the action has no effect (e.g. if the agent was heading North, it will be directed towards West)
4. `STOP_MOVING`: the agent remains in the same cell

5. `DO_NOTHING`: the agent performs the same action as the last time step

Usually, only a handful of the reported actions can be perfomed on a given cell, meaning that most of the times the actual number of choices is much less than 5 (we will address this issue in section 3.3).

Complications

The main complications of the Flatland challenge are given by speed profiles, conflicts and malfunctions. In particular, about speed profiles, each and every agent could have a different velocity. The standard speed (and the maximum one) is 1, which means that the agent crosses one cell in one time step. Speeds can have values in range $(0, 1]$: if an agent has speed s, it means that it needs $\lceil \frac{1}{s} \rceil$ time steps to transition from one cell to the next one.

Speeds are assigned to agents based on a custom probability mass function, so that $P(S = s)$ represents the probability that the speed S of an agent is equal to s. For example, we could have the following *pmf* (representing a uniform distribution over values $\{\frac{1}{4}, \frac{1}{3}, \frac{1}{2}, 1\}$):

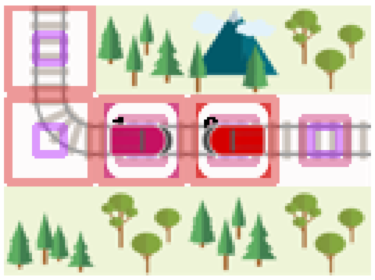
$$\begin{cases} P(S = s) = 0.25 & \text{if } s \in \{\frac{1}{4}, \frac{1}{3}, \frac{1}{2}, 1\} \\ P(S = s) = 0 & \text{otherwise} \end{cases} \tag{2.1}$$

Clearly, the speed factor is a critical one to observe when trying to minimize the total number of time steps in a multi-agent scenario: for example, different agents have to understand that faster ones should go first if a decision has to be made. This is because we would like to avoid bottlenecks of any kind inside the railway network and slower agents can definitely become an issue in an environment with relatively long sequences of straight rails.

About the second complications, i.e. malfunctions, agents can experience defects or failures which do not let them go on in their path towards the target. A malfunction in the Flatland environment is modeled by a Poisson distribution $1 - P_\lambda(n = 0) = 1 - \frac{\lambda^n}{n!} \cdot e^{-\lambda} = 1 - e^{-\lambda}$, where $\lambda \in (0, 1]$ is the malfunction rate s.t. $\frac{1}{\lambda}$ represents the mean frequency of occurrence of malfunctioning events. For example, if malfunctions are to be expected once every 80 time steps for an agent, then $\lambda = \frac{1}{80} = 0.0125$ and the probability of a malfunction in each time step is equal to $1 - e^{-0.0125} = 0.01$, while if malfunctions are more rare (e.g. once every 200 time steps), then we would have a probability value of $1 - e^{-0.005} = 0.004$. Once an agent is malfunctioning, it stays so for a random number of time steps, bounded by parameters indicating the minimum and maximum duration of a malfunction period.

Again, the malfunction factor is critical, in that it could prevent one or more agents from reaching their targets in the minimum number of time steps. In this way, transitions in the environment become stochastic, thus leading to a slower and much more difficult optimization procedure.

About the third complication, i.e. conflicts, we can define it as a state in which agents cannot perform any action, because they are "blocked" by one or more neighboring agents. Different conflicting situations can arise in practice, where the most basic one is given by two agents heading in different directions in a sequence of straight rails (see figure 2.5a): this situation can in turn cause other deadlocks to happen, as shown in figure 2.5b.


(a) Two agents

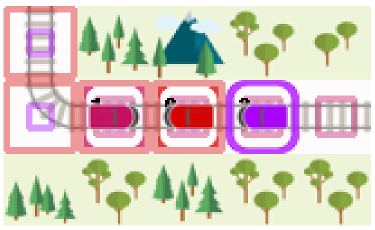

(b) Three agents

Figure 2.5: Examples of deadlocks

A full deadlock situation is a state in which every agent is in deadlock: in that case, no agent would be able to arrive at its target. These blocking situations are one of the most intricate and complex factors that should be addressed when designed autonomous agents in the Flatland environment, since they are quite frequent (and they get more frequent as the number of agents increases and the dimension of the grid decreases) and have disastrous consequences (at least in a real-world scenario).

Differently from malfunctions, deadlocks totally prevent agents from reaching their destination: when a malfunction occurs, agents that could previously reach their targets will still be able to reach them once the malfunction period is over, while when a conflict occurs, at least two agents will never arrive at their targets.

3 Environment

In this chapter we are going to explore better ways to encode the Flatland environment and how agents can interact with these representations.

3.1 Railway encoding

As already described in chapter 2, the default Flatland representation was not built with the goal of efficiency in mind, since it stores each and every cell of the 2D grid, while only `rail` cells are the ones that are actually used by agents. An alternative and more efficient representation could be to use some kind of sparse matrix implementation, where `empty` cells are not stored at all, but this would only be beneficial from the point of view of memory occupancy, while the usage of more specific data structures could also improve other aspects, like the computation of shortest paths from the agent's position to its target.

In this work, we decided to rely on a graph structure. In particular, we were inspired by the great work described in [5] where the author presents the so called Cell Orientation Graph (COG), in which nodes represent cells in the 2D grid as a triple (x, y, d), where (x, y) are the coordinates in the grid (origin in the top-left corner with x-axis looking right and y-axis looking down) and d is one of the four cardinal directions (representing the direction of entrance in the cell). In this way, one `rail` cell is represented by at most 4 nodes in the graph: this could seem like a major drawback, but in practice we can observe that the number of nodes is roughly equivalent to the number of cells in the corresponding grid, since we got rid of the `empty` ones.

Instead, edges in the graph are directed and represent legal transitions, so that no transition matrix or bitmap has to be stored in each node: it is simply encoded in the topology of the network. Moreover, the usage of directed edges greatly simplifies the computation of paths between nodes in the graph.

In order to further simplify the representation of the Flatland environment, we decided to entirely delete nodes which represented straight rails (with the exception of keeping straight rails containing targets and deadends). In this way, what we end up with is something that could be defined a Cell Orientation Junction Graph (COJG), since the remaining nodes are the ones in which an agent either has to make a decision or finishes its trip.

Because of the deletion of almost all nodes associated to straight rails, edges actually represent a connection between an interesting cell and another one (e.g. they link a junction to a target or a junction to a second junction). In order to maintain the same topology as before, the number of deleted straight rails between each pair of interesting nodes is used as the weight of the edge connecting them, so that the computation of shortest paths can automatically take that into account.

Figure 3.1 shows the comparison, in terms of visual representations, of the standard grid environment and both COG and COJG graphs, in a 32×16 map. In table 3.1 we show the improvements that can be gained by leveraging the usage of the Cell Orientation Graph, and in particular of its modified version, i.e. the Cell Orientation Junction Graph, by computing the number of nodes, edges and `empty` cells in each representation, in the same 32×16 map of figure 3.1.

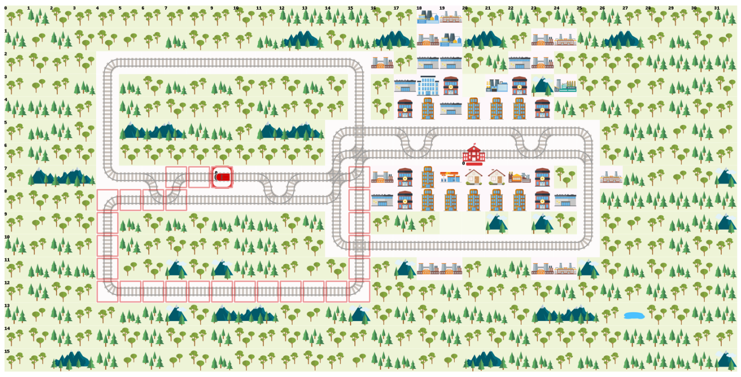
	Nodes	**Edges**	**Empty**
Grid	512	-	413
COG	226	254	0
COJG	78	106	0

Table 3.1: Grid, COG, COJG comparison in a 32×16 map

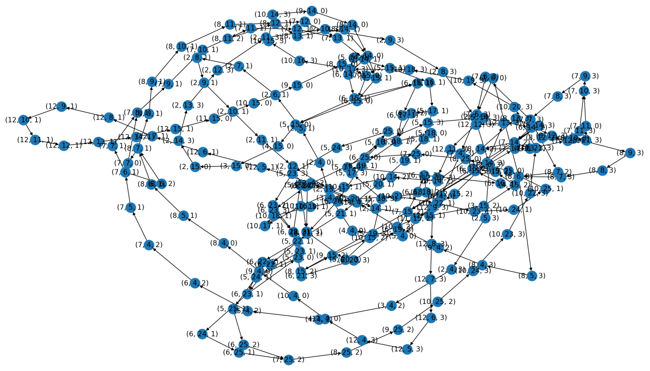
Since a $32x16$ map is too small to observe big improvements, in figure 3.2 we also report another example of a sparse grid environment of dimension 128×64 and we compare it with its encodings in table 3.2.

	Nodes	**Edges**	**Empty**
Grid	8192	-	7732
COG	1050	1096	0
COJG	136	182	0

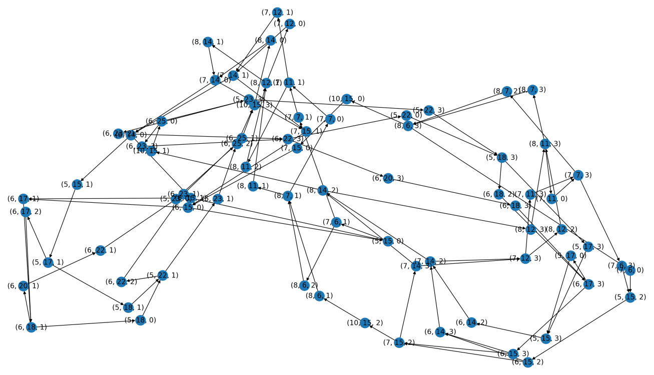
Table 3.2: Grid, COG, COJG comparison in a 128×64 map



(a) Grid

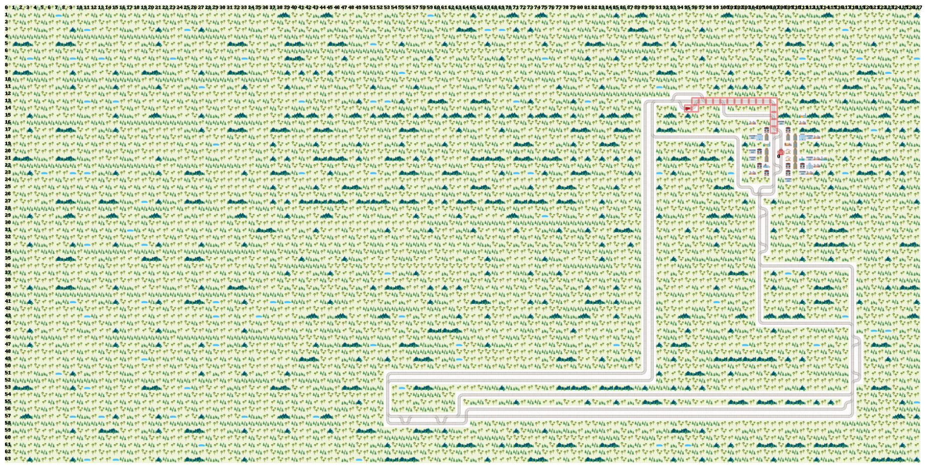


(b) COG



(c) COJG

Figure 3.1: Different railway encodings of a 32 × 16 map


(a) Grid

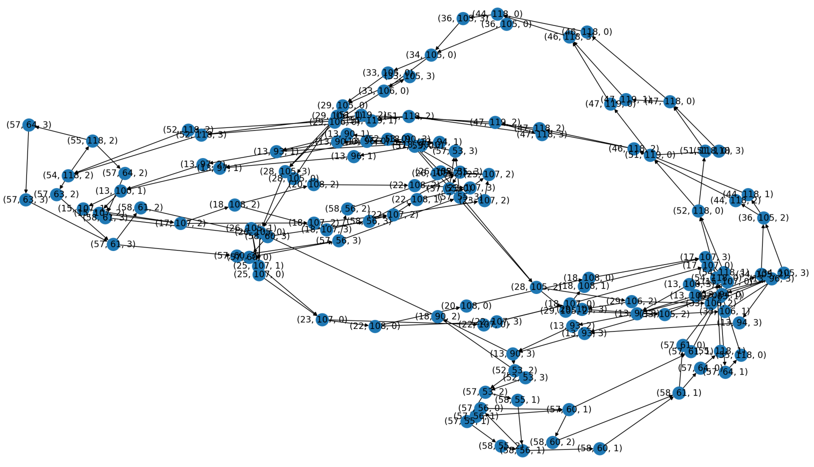

(b) COJG

Figure 3.2: Grid and COJG comparison in a sparse 128 × 64 map

3.2 Predictions

As reported in Flatland official FAQs, because railway traffic is limited to rails, many decisions that an agent has to take need to consider future situations and detect upcoming conflicts ahead of time. Therefore, Flatland provides the possibility of predictors that predict where agents will be in the future. The stock predictor simply assumes that each agent travels along its shortest path.

3.2.1 Shortest and deviation paths

One of the key advantages of COJG is that it enables us to apply standard shortest path algorithms on directed weighted graphs without any kind of overhead. In particular, we decided to use Dijkstra's algorithm since we do not have negatively weighted edges (in that case, Dijkstra's algorithm could get stuck in a cycle containing at least one edge with negative weight, since it may cycle an infinite number of times and sometimes go down with the total cost as much as it likes).

To build our predictor, we took inspiration from the standard predictor given by the Flatland library as a baseline, which computes shortest paths directly on the 2D grid representation. Our predictor instead takes into account two possibilities: an agent either follows its shortest path or deviates from it. The computation of shortest paths is incremental, meaning that once it is done it only gets updated (i.e. the first node is removed). The only event that causes a re-computation of the shortest path is when the agent makes a choice that does not follow the stored path.

Deviation paths are instead computed on the fly and they represent alternatives from each node of the shortest path. The computation of deviation path i, given the shortest path $n_1, \ldots, n_i, n_{i+1}, \ldots, n_d$, is simply another call to the Dijkstra's routine, where edge (n_i, n_{i+1}) is forbidden. In this example, d represents the maximum depth of both shortest and deviation paths.

In this way, the prediction becomes a list of paths, where the first one is the shortest path, composed by at most d nodes, and the following ones are $d - 1$ deviation paths, still composed by at most d nodes. When an agent is relatively close to its target (i.e. less than d nodes away from it), the actual path could have a dimension which is less than d: to avoid troubles when dealing with dynamically sized vectors, paths are padded to the full depth with special symbols.

In case an agent cannot reach its target anymore, the prediction only comprises a path of two nodes, i.e. the agent's position and the next node in the COJG graph. This mini-path is stored because predictions are used to compute possible deadlocks and, even if an agent cannot arrive at its target anymore, it is still present in the railway environment and it could be a possible source of conflicts.

3.3 Real decisions and choices

3.3.1 Real decisions

As already hinted in the previous sections, Flatland railway environments are structured in such a way that most of them are composed by long sequences of straight rails, where an agent does not have to commit to any thoughtful choice, but it simply has to follow the path on which it is positioned. The agent is instead required to make a decision in two simple cases: whenever it reaches a fork, i.e. a cell that has more than one successor, or immediately before a join, i.e. before entering another path that crosses its next cell.

Following the reasoning described in [5], we categorize as real decisions those times in which an agent has to select an action when it is positioned in a fork, before a join or in a combination of the two cases. An agent in a fork has to decide on which branch to go next (and ideally, it should not choose to stop on that cell), while an agent before a join has to decide if it wants to stop (maybe to let another agent pass through the join cell) or continue moving.

This idea of introducing real decisions in the Flatland environment greatly reduces the number of calls to the model that is tasked to map states to actions, since the number of real decisions to take is much lower than the number of default actions that one has to provide in the standard framework. In particular, the default number of actions to select is equal to the number of steps that the agent performs in the environment, while the number of real decisions to take is entirely disentagled from the steps parameter, since it is completely dependent on the structure of the environment (e.g. sparser railways tend to have less real decisions w.r.t. denser ones, given the same grid size).

The concept of real decisions leads to a modification in the structure of the COJG graph reported in section 3.1, so that both types of decisions are encoded as nodes. In particular, COJG is missing most nodes related to cells before a join, since they are usually part of straight rails. Currently, our implementation patches the COJG graph, but a cleaner choice would be to actually add the before join nodes from the start of the building process, which could be something to try for future improvements.

3.3.2 Choices

Further analysis related to decisions, again presented in [5], shows a smart action space reduction that could be beneficial to reinforcement learning models. In particular, Flatland uses a discrete action space composed of 5 actions, which can be reduced to a total of 3. From now on, we will refer to elements of this new action space as "choices".

Choices are mapped in such a way that they correspond to the left-most and right-most directions available to the agent at each time-step. In this way, our model only has to predict whether the agent has to follow the track to its left or the track to its right or simply stop. After a choice is selected, it is mapped back to the initial action space by looking at the type of cell the agent is currently on, to ensure compatibility with the Flatland environment.

The new action space is composed by the choices `CHOICE_LEFT`, `CHOICE_RIGHT` and `STOP`.

So, we have to specify two different mappings, the forward and the backward one. The action to choice mapping works as follows:

1. If the selected action is `MOVE_LEFT` and that action is legal from the current cell of the agent, then return `CHOICE_LEFT`
2. If the selected action is `MOVE_RIGHT` and that action is legal from the current cell of the agent, then
   (a) If only `MOVE_RIGHT` can be perfomed, then `CHOICE_LEFT` is returned
   (b) Otherwise, `CHOICE_RIGHT` is returned
3. If the selected action is `MOVE_FORWARD` and that action is legal from the current cell of the agent, then
   (a) If `MOVE_LEFT` can be perfomed, then `CHOICE_RIGHT` is returned
   (b) Otherwise, if `MOVE_RIGHT` can be perfomed, then `CHOICE_LEFT` is returned
   (c) Otherwise, the last resort is `CHOICE_LEFT`
4. If `STOP_MOVING`, we have a one-to-one mapping with the `STOP` choice, so that the agent decides to stand still

Instead, the choice to action mapping works as follows:

1. If `CHOICE_LEFT`, then priorities are `MOVE_LEFT`, `MOVE_FORWARD`, `MOVE_RIGHT`
2. If `CHOICE_RIGHT`, then priorities are `MOVE_RIGHT`, `MOVE_FORWARD`
3. If `STOP`, then the only possible outcome is `STOP_MOVING`
4. Otherwise, last resort is `DO_NOTHING`

In the previous bullet points, the word "priority" represents the sequential way in which actions should be selected. For example, if a `CHOICE_RIGHT` has to be mapped back and `MOVE_RIGHT` is a legal action from the current cell of the agent, then that action would be selected, otherwise `MOVE_FORWARD` would be chosen (again, if legal).

3.4 Observations

An observation is a representation of the environment in a particular state. Ideally, this snapshot should be rich enough to convey the right amount of information (so as to distinguish between different configurations), but also small enough that it would be efficient to compute at any given time-step. An observation should also be able to integrate information from local (in our case a train) and global (in our case the grid and all the agents) perspectives.

Flatland provides different observators out of the box (the ones roughly depicted in figure 3.3). In particular, the implemented global observation is composed of the following elements:

- Transition map tensor with shape $(h, w, 16)$, assuming 16 bits encoding of transitions
- A tensor with shape $(h, w, 2)$, containing the position of the given agent's target in the first channel and the positions of the other agents' targets in the second channel (flag only, no counter)
- A tensor of shape $(h, w, 5)$ with the following channels:
    1. Agents position and direction
    2. Other agents positions and direction
    3. Agents malfunctions
    4. Agents fractional speeds
    5. Number of other agents ready to depart

The implemented local observation should instead gather information of the rail environment around the given agent. This observation is composed of the following elements:

- Transition map tensor of the local environment around the given agent, with shape $(v_h, 2 \cdot v_w + 1, 16)$, assuming 16 bits encoding of transitions
- A tensor with shape $(v_h, 2 \cdot v_w + 1, 2)$ containing the target position of the given agent (if in the agent's vision range) in the first channel and the positions of the other agents' targets (again, if in the agent's vision range) in the second channel
- A tensor $(v_h, 2 \cdot v_w + 1, 4)$ containing the one-hot encoding of directions of the other agents at their position coordinates (if in the agent's vision range)
- A 4 elements array with a one-hot encoding of the current direction of the given agent

Here, the parameters v_w and v_h define the rectangular view of the agent (of dimension $(2 \cdot v_w + 1) \times v_h$) around its position. One thing to notice about the standard local observation is that it does not contain any clues about the target location, if it's out of range: thus, navigation on maps where the radius of the observation does not guarantee a visible target at all times will become very difficult.

Moreover, Flatland discourages the use of both the given global and local observations, since the first one does not seem to encode enough information to be effective in a multi-agent scenario, while the second one is simply deprecated in the newest version of their Python library, which makes it practically useless. Because of this, more advanced observations must be used in order to achieve good results on a wide range of environments. In particular, in the following sections we will take a look at another observation present in Flatland library, i.e. the tree observation, and we will use it as the starting point to build our custom binary tree observation, which also integrates all the knowledge and findings that we described in the previous chapters.

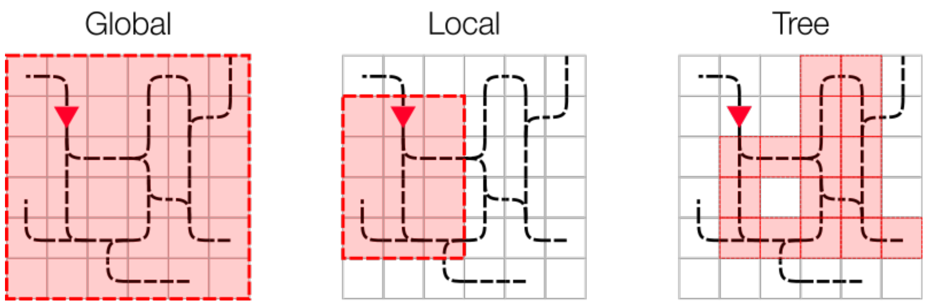


Figure 3.3: Default observators

3.4.1 Tree

The tree observation is the last one provided out of the box in the Flatland package and it's definitely the most effective overall (w.r.t. the already mentioned global and local ones). This observator builds a tree with arity 4, s.t. the children of a node always follow a left, forward, right and backward ordering. In particular, a node in the tree represents a deadend, a switch, a target destination or a blank (meaning that no information is carried by that specific node). The children of a node represent its successor nodes when following the left/right/forward branch on a switch or the backward one on a deadend. The mentioned movements are sorted relative to the current orientation of the agent, rather than using standard transition directions (i.e. the four cardinal directions). Whenever a node does not have one of the mentioned successors, that branch becomes useless and gets filled with padding values.

Figure 3.4 shows a mapping between a state and the corresponding tree observation, from the point of view of the agent represented by the red triangle. The colors in the figure illustrate what branch the cell belongs to (red means left, pink means forward, green means right and blue means backward). If there are multiple colors in a cell, then this cell will be present multiple times in the tree. The symbols in the figure have instead the following meaning:

- Cross: no branch was built, i.e. the parent does not have a valid transition to represent the child
- No children: the node is a terminal node (i.e. a deadend, a cell without possible transitions or a leaf of the tree, indicating that the maximum depth was reached)
- Circle: a node filled with information

Whenever a node is valid, it will be packed with the following features:

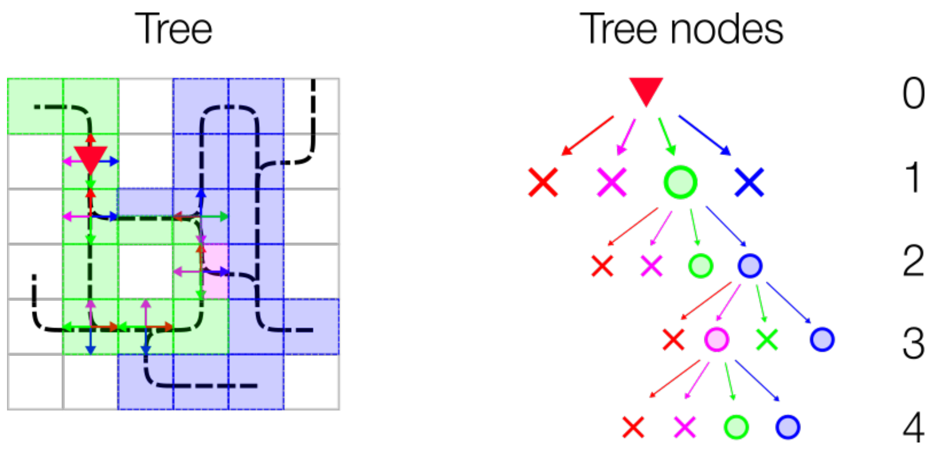


Figure 3.4: Tree observator

1. If the target of the given agent lies on the explored branch, the current distance from the given agent (in number of cells) is stored
2. If another agent's target is detected, the distance in number of cells from the given agent's current location is stored
3. If another agent is detected, the distance in number of cells from the given agent's position is stored
4. If another agent predicts to pass along this cell at the same time as the given agent, the distance (in number of cells) from the given agent's position is stored
5. If an unusable switch is detected for the given agent, the current distance (in number of cells) from the given agent's position is stored
6. Distance in number of cells to the next branching
7. Minimum distance from the node to the given agent's target, considering the current direction of the agent
8. Number of agents present in the same direction as the given agent
9. Number of agents present in the opposite direction as the given agent
10. Number of time steps that the given agent remains blocked, if a malfunctioning agent is encountered
11. Slowest observed speed of agents in the same direction as the given agent
12. Number of agents ready to depart, but not yet active

One key issue related to the described tree observation is related to both time and memory efficiency. In particular, we mentioned a maximum depth parameter that controls how far the observation should reach, in terms of consecutive switches, targets and deadends visited. Clearly, since each node has

4 children and the output of the observator should be consistent, running it requires to build a complete tree, which we know to have a number of nodes equal to:

$$n = \sum_{i=0}^{d-1} a^i,$$

where d is the height of the tree (i.e. our maximum depth) and the arity a is equal to 4. Just to make a quick example, if $d = 7$ then $n = 4^0 + 4^1 + 4^2 + 4^3 + 4^4 + 4^5 + 4^6 = 5461$. In this way, in the worst case, for each time-step we would have to examine $t \cdot n$ nodes, where t is the number of trains, since one observation is needed for each agent. Following the previous example, in a scenario with $t = 10$ agents and a tree observation with depth $d = 7$, the reported features should be computed for each of the $5641 \cdot 10 = 56\,410$ analyzed nodes at every time-step.

3.4.2 Binary tree

In order to exploit what we've built so far and have an hopefully more efficient computation of each observation, a new observator was created, starting from the same notions described in section 3.4.1.

In particular, our observator builds a feature tensor of shape (d, d, f), where d is the maximum number of hops in the COJG graph to consider and f is the total amount of features for each node. This tensor contains the features of the nodes in the shortest path as the first row and the features of the nodes in the deviation paths (which are exactly $d - 1$) as the following rows.

Each node has the following features:

1. Number of agents (going in the same direction as the one of the given agent) identified in the subpath from the root up to each node in the path
2. Number of agents (going in a direction different from the one of the given agent) identified in the subpath from the root up to each node in the path
3. Number of malfunctioning agents (going in the same direction as the one of the given agent) identified in the subpath from the root up to each node in the path
4. Number of malfunctioning agents (going in a direction different from the one of the given agent) identified in the subpath from the root up to each node in the path
5. Minimum distances from an agent to other agents (going in the same direction as the one of the given agent), in each edge of the path
6. Minimum distances from an agent to other agents (going in a direction different from the one of the given agent), in each edge of the path
7. Maximum number of malfunctioning turns of other agents (going in the same direction as the one of the given agent), in each edge of the path

8. Maximum number of malfunctioning turns of other agents (going in a direction different from the one of the given agent), in each edge of the path
9. Distances from the target, from each node in the path
10. Path weights (in number of turns) to reach the given node from the root one
11. Number of agents using the node to reach their target in the shortest path
12. Number of agents in deadlock in the previous path, assuming that all the other agents follow their shortest path
13. How many turns before a possible deadlock
14. If the node is a fork or not
15. How many turns the current agent has been repeatedly selecting the stop action

Initially, we tried to use the feature tensor as our final observation. In particular, we linearized it as a $d \times d \times f$ vector and fed it into one of the neural models that will be described in the next chapters. In this way, the model was not converging to reasonable solutions, since we were ignoring consistency, meaning that in neural models each and every input neuron should always represent the same measure. Instead, since our feature tensor was built by simply considering paths in the environment, one specific node could have represented either the left or the right branch being taken from the previous node, in different runs. This was definitely confusing the agents, so that no convergence was reached.

Because of this, we resorted to the same idea that was presented in 3.4.1, but, thanks to the new action space introduced in 3.3.2, we were able to shape the feature tensor like a binary tree (i.e. a tree with arity $a = 2$), where branches identify choices, i.e. `CHOICE_LEFT` or `CHOICE_RIGHT`. One key difference that our custom binary tree has w.r.t. the standard tree observation is the injection of prior knowledge: this is done by filling in values of features that are only related to nodes present either in the shortest or in one of the deviation paths. In this way, the binary tree contains only information about paths that could lead to the agent's target, by completely disregarding the ones that could not.

Since the resulting tree has half the arity of the one given by Flatland and we still need a complete tree for consistency reasons, let's consider again the example reported in 3.4.1 to roughly check for efficiency improvements, where a maximum depth of $d = 7$ was chosen, with $t = 10$ trains. Our complete binary tree would have $n = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 = 127$ nodes (compared to 5641) and the number of total nodes for which features would have to be computed at each time-step would be around $127 \cdot 10 = 1270$ (compared to 56 410). Moreover, the reported number (1270) is an upper bound on the actual number of nodes that should be analyzed, since different time-steps will require different agents to compute an observation, i.e. only those agents that find themselves in a real decision cell.

To sum up efficiency improvements, each feature is built from scratch, by leveraging the performance gains obtained with the COJG graph. Moreover, the observation needs to be computed only when the agent faces a real decision (as described in section 3.3.1), since in every other scenario the action that the agent should take is implicit. By combining this with the much lower size of the tree (compared to Flatland's implementation), we are able to compute more observations with less resources overall. Table 3.3 shows a comparison of the standard tree observation and our custom binary tree one, in terms of both running times and memory occupancy for a single observation. As we can see, the binary tree observation is around 6 times faster on average and occupies 30 times less memory on average than the tree one.

	Time (s)		**Memory (MB)**	
	Avg	**Std**	**Avg**	**Std**
Tree	0.1978	0.0199	0.4806	0.0
Binary tree	0.0335	0.0124	0.0153	0.0

Table 3.3: Tree vs. binary tree efficiency in a 48×27 map, with $d = 7$

3.4.3 Normalization

We already mentioned that an observation should be considered as a snapshot of a state from the point of view of an agent; then we reported that this snapshot should be consistent across different runs and that it should also be shaped as a tensor of fixed size. All of this pre-preprocessing is a required step that should be carefully considered when working with neural models (the ones that will be described in subsequent chapters).

Another important aspect about neural networks is that they mostly require input features to be in comparable ranges: this can be achieved by either normalization or standardization. To be clear, the input features that we are talking about are the ones associated to elements of an observation (e.g. a node in our binary tree).

About standardization, a feature x is mapped to $x_s = \frac{x-\mu_x}{\sigma_x}$, where μ_x and σ_x are the mean and standard deviation of feature x. In this way, the resulting x_s would have zero mean and standard deviation equal to 1. One thing to notice, though, is that it requires either to know the underlying distribution of feature x and its corresponding parameters (μ_x and σ_x) or a sample of examples from which parameters could be estimated. In the case of reinforcement learning, we definitely do not have pre-computed samples, since we work in an online fashion, meaning that observations are computed on the fly at each time-step and the network gets called with a batch of previously collected observations. Because of this, the only way in which we could standardize features would be to either know the underlying distribution of each one (which is mostly not the case), or use normalization layers in the network itself (e.g. batch normalization [4] or layer normalization [2]).

About normalization, it could be used to refer to many different things, but the main idea is again to translate an input range $[l, u]$ to a fixed output range $[min, max]$ (e.g. $[0, 1]$ or $[-1, 1]$). Usually, when the input is translated into $[0, 1]$ we talk about normalization, while when the input is translated into a range different than $[0, 1]$ we talk about min-max scaling. The general formula of min-max scaling is $x_n = l + \frac{x-min}{max-min} \cdot (u - l)$. In the more specific case of default normalization, the formula becomes $x_n = \frac{x}{max}$.

Normalization enables us to exploit domain knowledge in order to infer ranges for each feature of our observation. In particular, distances are normalized by considering a fixed observation radius as maximum and either zero or the negated radius as minimum (in the case of negative distances, which we used to indicate agents behind the given one, for example), while other features have much clearer ranges (e.g. features related to malfunctions have a minimum at 0 and a maximum at the maximum duration of a malfunction, which is a parameter of the environment).

4 Reinforcement learning setting

Introduction

In a single-agent standard reinforcement learning scenario (see figure 4.1), the problem is modeled as an MDP (Markov Decision Process), where at each discrete time-step t the agent observes the environment's state s_t (where $s_t \in S$ and S is the whole state space) and takes an action a_t (where $a_t \in A$ and A is the whole action space) that brings it to a new state s_{t+1}. Moreover, the agent receives a reward r_{t+1} proportional to the quality of taking action a_t in state s_t. Usually, rewards are local, meaning that only the given (s_t, a_t) pair should be considered when computing them. Then, the agent should learn a policy, i.e. a function $\pi : S \times A \rightarrow [0, 1]$, that maximizes the expected cumulative return, i.e. $R = \sum_{t=1}^{\infty} \gamma^t r_t$, where $\gamma \in (0, 1)$ is the discount rate (a parameter that intuitively encodes how much importance should be given to immediate or future rewards, s.t. $\gamma \rightarrow 0$ favors the former while $\gamma \rightarrow 1$ favors the latter).

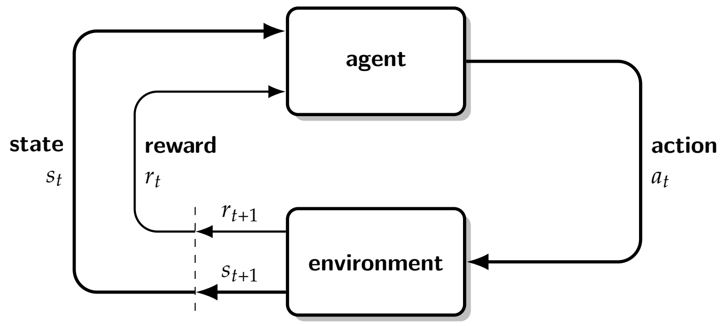


Figure 4.1: Single agent RL scenario

Since Flatland is actually a multi-agent environment, the formal definition of the reinforcement learning scenario slightly changes, but the main ideas still hold. As reported in [8], MARL (Multi-Agent Reinforcement Learning) typically represents the environment as a stochastic game. While the name is different, its fairly similar to an MDP. States become joint states of all the agents, with different rewards corresponding to each possible joint action. Transition functions remain analogous to the single-agent case, replacing states and actions accordingly.

One of the main challenges of MARL is related to non-stationary transitions, i.e. given a distinct state and action pair, the transition probabilities to other states are not constant (state transitions become dependent on the joint actions of all the agents).

Value functions

As reported in [10], almost all reinforcement learning algorithms involve estimating value functions, i.e. functions of states (or of state-action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state). The notion of how good here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return. Of course the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular ways of acting, called policies (which we already introduced in the previous section).

Formally, the value function of a state s under a policy π, denoted $V_\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs, we can define V_π formally by

$$V_\pi(s) = \mathbb{E}_\pi \left[G_t \mid s_t = s \right],$$

where $G_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$ is the discounted return at time-step t and $E_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π.

Moreover, we can also define the value of taking action a in state s under a policy π, denoted $Q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid s_t = s, a_t = a \right],$$

The relationship between the V and Q functions is as follows:

$$V_\pi(s) = \sum_a \pi(s, a) \cdot Q_\pi(s, a)$$

Optimal policy

The main goal of reinforcement learning approaches is to find an optimal policy $\pi^* = \arg\max_\pi R$. To do so, different types of algorithms could be employed:

- Model-based: the agent tries to understand the world and create a model to represent it, so that, after learning, it could make predictions about what the next state and next reward will be before taking an action
- Model-free: they rely on real samples from the environment and never use generated predictions of next state and next reward to alter behaviour
  - Value-based: the policy is implicit, so that at each time-step the agent chooses the action that brings it to the next state which has the best evaluation
  - Policy-based: it avoids forming both the MDP and the value functions, and instead tries to directly learn the (optimal) policy

In this work we mainly focused on model-free, value-based techniques. Their goal is to implicitly find the optimal policy, by sequentially pushing the related value functions towards higher quality ones. In this regard, the final aim of our RL agents is to find the optimal action-value function (also called Q function), by leveraging Bellman's optimality equation:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[G_t \mid s_t = s, a_t = a, \pi\right]$$

4.1 Rewards and shaping

Designing rewards is a key property for the success of reinforcement learning agents. Rewards should be simple and straightforward, but they should also encode the right amount of information extracted from the domain of the application. Hence, they cannot be designed in a general fashion, because they are problem-dependent.

Flatland provides the following rewards out of the box:

- -1 for each step taken
- +1 when every agent arrives at their target

Training agents with the standard reward didn't seem effective in a lot of ways. For example, from our point of view it doesn't make sense to give an agent a reward of +1 only if all of them reached their targets. A more sensible idea would be to kind of even out all the negative rewards that the agent got during their journey to the target, but also leaving information about the number of steps that it took to arrive at the destination.

Backing the reported reasoning, the following rewards shaping was perfomed:

- If the agent has arrived to its target, a positive reward equal to the maximum number of steps minus the current number of steps is returned (in this way, if an agent arrives in n_1 steps in one execution and in $n_2 \leq n_1$ steps in another execution, information about the number of steps is not lost and the agent should learn to prefer the transitions that made it receive a reward of n_2)
- If the agent is in deadlock, a reward equal to the negated maximum number of steps is returned
- Because of the introduction of real decisions, rewards have to be accumulated whenever the agent is not in a decision cell (so that, for example, if an agent travels along a sequence of n straight rails, its reward would be equal to $-n$ instead of -1 when it exits the sequence)
- If an agent performed a stop action, give the agent a reward which is worse than the the reward it could have received by choosing any other action. Then, multiply this weight by the number of turns the agent consequently decided to perform a stop action and compute the product of it with a stop penalty parameter. Finally, this reward is clipped to

the maximum value given by a deadlock penalty. In this way, performing a lot of consecutive stop actions has an exponential decrease in the received rewards.

4.2 Action selection

A reinforcement learning algorithm could be considered as a search algorithm in the huge space of feasible policies. In that case, the aim of our agent would be to travel along the landscape of "policy values" in order to find the global optimum, i.e. the best policy overall. The final goal would thus be to constantly improve the current best policy, via some kind of local search, which in the case of neural networks is implemented using gradient descent.

Because of this local search perspective, another key aspect of the reinforcement learning algorithm is to balance the exploration-exploitation tradeoff. In particular, by exploration we mean probing a large portion of the search space with the hope of finding other promising solutions that are yet to be refined, while exploitation is instead related to the refinement of an already promising solution, by intensifying the search in its neighborhood.

In practical terms, at each time-step an action selector has to decide if the agent should follow the action that our current policy marks as the best one (exploitation) or choose another action (exploration), from the set of available ones.

Moreover, the farther we are in the training process of the RL algorithm the lower should be the exploration parameter, meaning that a decaying mechanism should be implemented to decrease the exploration value at each time-step. Different types of decaying methods could be implemented, like an exponential or a linear one (our experiments were conducted mostly with a linear decay, with a pre-defined number of exploration episodes).

4.2.1 ϵ-greedy

The ϵ-greedy action selector is a standard one in the RL community. It works by treating the ϵ parameter (which should be in range $[0, 1]$) as the probability of exploration, so that $1 - \epsilon$ represents the probability of exploitation.

In practical terms, given a random number $r \in [0, 1]$ sampled from a uniform distribution and the set of possible actions A, the action a_t that is selected at time t is the following:

$$a_t = \begin{cases} \max_{a \in A} Q_{\pi_t}(s_t, a) & \text{if } r \geq \epsilon \\ \texttt{random\_choice}(A) & \text{if } r < \epsilon \end{cases},$$

where `random_choice` is a routine that samples from the given set, with a uniform distribution.

By tweaking a little bit the ϵ-greedy selector we could also get two other selectors:

- Greedy: set $\epsilon = 0$ and avoid decaying it (always select the best action)
- Random: set $\epsilon = 1$ and avoid decaying it (always select a random action)

4.2.2 Boltzmann

The Boltzmann action selector is less common in the RL community. It works by computing a modified softmax of the values associated to actions and using the resulting probability distribution to sample an available action. The balance between exploration and exploitation is controlled by the temperature parameter τ (which should be in range $[0, 1]$), so that higher values of τ will result in higher exploitation.

In practical terms, we have to model the probability of each action a_i being selected at time t with the following equation:

$$P_\tau(a_i) = \frac{exp(Q_{\pi_t}(s_t, a_i)/\tau)}{\sum_a exp(Q_{\pi_t}(s_t, a)/\tau)}$$

Then, a sample is drawn from the set of actions A, so that the probability of selecting action i is equal to the previously computed value $P_\tau(a_i)$.

By tweaking a little bit the Boltzmann selector we could also obtain the so-called Categorical action selector, where $\tau = 1$ and no decaying is applied to it. In this way, the probability of choosing an action is given by a default softmax over the obtained Q function values.

4.3 Action masking

One important problem that we faced while implementing our agents was related to differenting legal and illegal moves in the learning process. In particular, in the Flatland environment the set of available choices at each time-step is strictly related to the position of the agent: for example, an agent in a straight rail can only decide to follow the path over which it is placed (so only one action would be legal), while an agent that finds itself in a fork would have more than one available actions to choose from.

Computing legal and illegal actions is not a problem in the Flatland environment, since it is fully observable, but injecting this knowledge in the reinforcement learning scheme, so that the agent can learn to avoid illegal moves, is not straightforward. A simple way to do so would be to add a penalty term in the reward, but in this way you are not completely guaranteed that agents will never select an illegal action, because that step would be part of the learning process (i.e. agents would have to learn to avoid those actions).

A more all-round solution would instead completely prevent agents to pick a move that is not available (i.e. agents should directly ignore those actions). To do so, we decided to mask out illegal moves whenever the algorithm has to do some kind of computation or selection on the set of actions. Specifically, in the action selectors we replaced every instance of the max, $\arg\max$ and $\operatorname{soft}\max$

operator with their ad-hoc masked counterparts and we did the same when computing Bellman optimality.

5 DQN

A DQN (Deep Q-Network), as was first presented in [7], leverages neural networks from the point of view of universal function approximators, to estimate the optimal Q^* state-action function.

The main idea behind DQN is to estimate Q^* with Q_θ and use the Bellman equation to update the weights θ of the network so that $Q_\theta \to Q^*$. In particular, we will use a regression loss (e.g. MSE or Huber) like the following to minimize the so-called TD (Temporal Difference) error:

$$L(\theta) = \mathbb{E}_{(s_t, a_t, r_{t+1}, s_{t+1})}\left[\left(\underbrace{r_{t+1} + \gamma \max_{a_{t+1}} Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t)}_{TD}\right)^2\right]$$

About the network architecture, the literature does not follow a clear definition, so that the term DQN is mostly linked to the whole training pipeline required to reach good estimates of the optimal Q function. For example, in the original paper [7] DeepMind researchers use an initial CNN (Convolutional Neural Network), since their input is represented by videogame frames, followed by an MLP (Multi-Layer Perceptron). In our case, we sticked to just using the final fully-connected layers, as shown in figure 5.1, where the number of input neurons is given by the linearization of the selected observation, the number of hidden cells and the number of hidden layers are hyperparameters, while the number of outputs is equal to the size of the action space (in our case we have exactly 3 choices).

Another thing to notice is that the weights used for the target versus the current (presumably non-optimal) Q values are different. In practical terms, a separate network is used to estimate the TD target. This target network has the same architecture as the function approximator but with frozen parameters. Every T steps (a hyperparameter) the parameters from the Q-network are copied to the target network. This leads to more stable training because it keeps the target function fixed (for a while). Because of this, assuming θ_i to represent the current network weights and θ_i^- the target network weights, the loss function is updated as follows:

$$L(\theta_i) = \mathbb{E}_{(s_t, a_t, r_{t+1}, s_{t+1})}\left[\left(r_{t+1} + \gamma \max_{a_{t+1}} Q_{\theta_i^-}(s_{t+1}, a_{t+1}) - Q_{\theta_i}(s_t, a_t)\right)^2\right]$$

There are two main ways two copy the weights from the current to the target network:

1. Hard update: every T steps, the current network is cloned into the target network

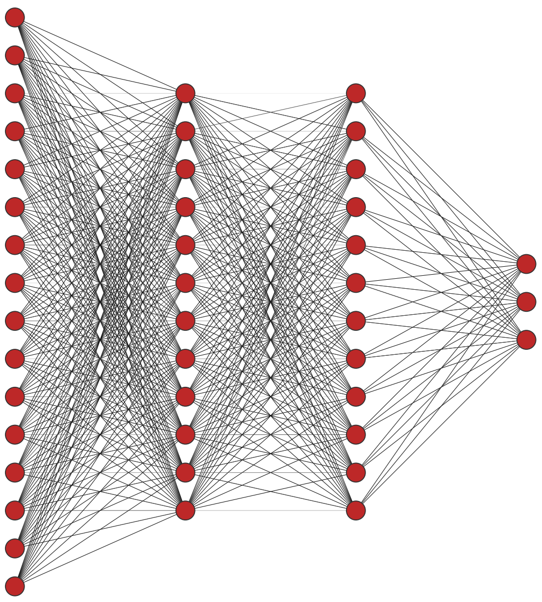


Figure 5.1: Simple DQN example with FC layers

2. Soft update (or Polyak average): at every time-step a portion of the current weights are shifted onto the target ones, so that at step i we have $\theta_i^- = \tau \cdot \theta_i + (1 - \tau) \cdot \theta_i^-$, where τ is an hyperparameter that controls the weight transfer amount

5.1 Experience replay

We already mentioned the problem of non-stationarity related to MARL, when agents have to deal with a stochastic environment, but in practice that same problem is also present in simpler RL algorithms, since the data distribution still changes as the agent learns new behaviours (which is in contrast with the standard assumption of a fixed underlying distribution of deep learning methods). Moreover, we know that supervised learning algorithms assume samples in a dataset to be *i.i.d.* (independent and identically distributed), so that if we model the RL problem as a supervised one (e.g. a DQN can be framed as a regression problem that minimizes the distance from the optimal Q function) we break the independence assumption since sequences of states are usually highly correlated.

To alleviate the problems of correlated data and non-stationary distributions, in the original DQN formulation [7] DeepMind researchers report the use of a data structure named experience replay (also known as replay buffer in the literature), which randomly samples previous transitions, and thereby smooths the training distribution over many past behaviors. In this way, the $(s_t, a_t, r_{t+1}, s_{t+1})$ tuple shown in the previous loss equations should be considered as a sample extracted from the current replay buffer.

In practical terms, there are two main ways to implement an experience replay E:

- Uniform [7]: previous experiences are drawn from a uniform distribution $\sim U(E)$
- Prioritized [9]: authors propose to more frequently replay transitions with high expected learning progress, as measured by the magnitude of their TD error

5.2 Improvements

5.2.1 Double

One important issue of standard DQNs is that they tend to overestimate Q values (which is mostly due to the max operator in the Bellman optimality equation), leading to unstable training and low quality policies.

In order to mitigate the issue, in [3] they propose to decouple the selection of an action from its evaluation. From a practical point of view, we only have to change how the TD error is computed:

$$L(\theta) = \mathbb{E}_{(s_t,a_t,r_{t+1},s_{t+1})}\left[\left(r_{t+1} + \gamma Q_{\theta^-}(s_{t+1}, \arg\max_{a\in A} Q_\theta(s_{t+1}, a)) - Q_\theta(s_t, a_t)\right)^2\right]$$

Intuitively, in double DQNs we are mitigating the overestimation bias by adding a second estimator, so that chances of both estimators being overoptimistic at the same action are simply lower. In this way, estimator Q_θ would "obtain the best action", while Q_{θ^-} would evaluate the Q values for the estimated action, so as to average out the inherent noise.

5.2.2 Dueling

The main goal of dueling architectures, as first presented in [11], is still to solve the overestimation problem described in 5.2.1. Moreover, since for most applications it seems unnecessary to know the value of each action at every timestep, the dueling architecture can learn which states are valuable and which are not, without having to learn the effect of each action for each state.

In particular, authors rely on the fact that the Q function can be decomposed in the sum of two functions, the value function V and the advantage function A, so that $Q_\pi(s,a) = V_\pi(s) + A_\pi(s,a)$. Moreover, from the expression of the state-value function $V_\pi(s) = \mathbb{E}\left[Q_\pi(s,a)\right]$ it follows that $\mathbb{E}\left[A_\pi(s,a)\right] = 0$.

From a practical point of view, a dueling DQN can be implemented by separating the Q function estimation into two streams, the "value" and "advantage" one (as shown in the last layers in figure 5.2), to then recombine them into a single output which represents the Q estimates themselves.

The recombination layer cannot simply be the sum of the presumed "value" and "advantage" outputs, since they actually aren't reliable estimates of the mentioned functions and they suffer from lack of identifiability (meaning that, given a Q value we are not able to recover the corresponding V and A values uniquely), as thoroughly described in [11]. To this end, authors decided to force the advantage function estimator to have zero advantage at the chosen action, so that output merging could be computed by the following formula:

$$Q(s,a) = V(s) + \left(A(s,a) - \max_{a' \in A} A(s,a') \right)$$

An alternative formulation of the recombination formula is to use the average (instead of the maximum) over the advantage function. In this way, the original semantics of V and A are lost, but authors report that it eases optimization, while still preserving the relative rank of the A (and hence the Q) values. Formally:

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

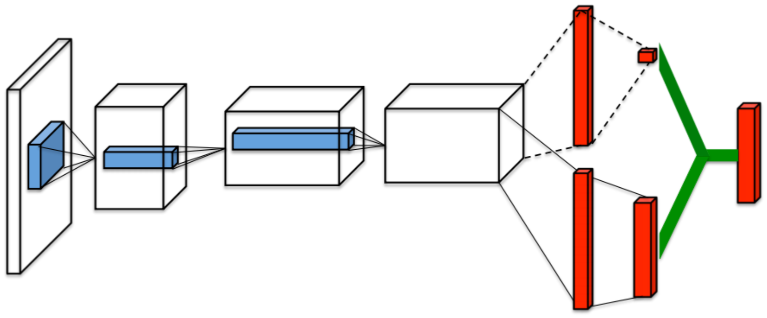


Figure 5.2: Dueling DQN architecture

5.2.3 Gradient noise reduction

Mnih et al. observed that large gradients can be detrimental to the optimization of DQNs. This was noted in the context of large reward values, but it is possible that overestimation bias can also introduce variance in the gradient and have a negative effect.

To solve this issue, in [7] authors decided to clip gradient values to a fairly small range (fixed to be $[-1, 1]$), while in [9] they went for a solution which clips the norm of the gradient to a specific maximum value (which was chosen to be 10).

By further inspecting the rationale behind Mnih et al. (quoted below), we can conclude that using an MSE loss and clipping gradient values to be in $[-1, 1]$

should give the same results as using an Huber loss without any kind of gradient noise reduction. However, it's not clear if Schaul et al. followed the same reasoning.

> *We also found it helpful to clip the error term from the update to be between* 1 *and* 1*. Because the absolute value loss function* $|x|$ *has a derivative of* 1 *for all negative values of* x *and a derivative of* 1 *for all positive values of* x*, clipping the squared error to be between* 1 *and* 1 *corresponds to using an absolute value loss function for errors outside of the* $(1, 1)$ *interval. This form of error clipping further improved the stability of the algorithm.*

5.2.4 Softmax Bellman

6 GNN

7 Results

8 Conclusions

Bibliography

[1] AIcrowd. *AIcrowd*. URL: https://www.aicrowd.com/.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML].

[3] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI'16. Phoenix, Arizona: AAAI Press, 2016, pp. 2094–2100.

[4] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 448–456. URL: http://proceedings.mlr.press/v37/ioffe15.html.

[5] Wälter Jonas. "Existing and novel Approaches to the Vehicle Rescheduling Problem (VRSP)". MA thesis. HSR Hochschule für Technik Rapperswil, Mar. 2020.

[6] Jing-Quan Li, Pitu B. Mirchandani, and Denis Borenstein. "The vehicle rescheduling problem: Model and algorithms". In: *Networks* 50.3 (2007), pp. 211–229. DOI: https://doi.org/10.1002/net.20199.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. "Playing Atari with Deep Reinforcement Learning". In: (2013). cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013. URL: http://arxiv.org/abs/1312.5602.

[8] Austin Nguyen. *Multi-Agent Reinforcement Learning: The Gist*. URL: https://medium.com/swlh/the-gist-multi-agent-reinforcement-learning-767b367b395f.

[9] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. *Prioritized Experience Replay*. cite arxiv:1511.05952Comment: Published at ICLR 2016. 2015. URL: http://arxiv.org/abs/1511.05952.

[10] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[11] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. "Dueling Network Architectures for Deep Reinforcement Learning". In: *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.