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)

September 11, 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 the available **transitions**: there are 16 different transitions in flatland, since there are 4 different orientations, and 4 other directions of exit from the cell. Thus, each cell is described by 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, only a maximum of 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 described with an integer `[0=North, 1=Est, 2=South, 3=West]`
- **Movement**: an integer describing the status of the agent `[0=READY_TO_DEPART, 1=ACTIVE, 2=DONE, 3=DONE_REMOVED]`

Since every agent is liable to malfunctions, much like real trains, there are properties to keep track of that in addition to 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 will occour
- **Number of malfunctions**: the total number of malfunctions for this agent
- **Max speed**: a fraction between 0 and 1. (*e.g.* 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 the transitions of the current cell allows it, the agent change it's direction toward the left, otherwise the action is masked with a `MOVE_FORWARD`. If the agent is not moving then update also it's state to moving.
- **MOVE FORWARD (2)**: It updates the direction of the agent following the transitions allowed in the current cell. If the cell is a dead-end the new direction is the opposite of the current. If the agent is not moving then update also it's state to moving.
- **MOVE RIGHT (3)**: If the transitions of the current cell allows it, the agent change it's direction toward the right, otherwise the action is masked with a `MOVE_FORWARD`. If the agent is not moving then update also it's state to moving.

- **STOP MOVING (4)**: Stop the agent in the current occupied cell and cause the change of value of the variable describing the state of the agent.

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*: calculated as 1 * beta, a sort of default penalty.
- *stop penalty*: is currently set to 0, penalty for stopping a moving agent
- *start penalty*: 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 (*i.e.* the *speed* variable). There are different rewards for different situations:

- agents that are in status `DONE` or `DONE_REMOVED` have zero reward.
- all agents that have finished in this episode (checked at the end 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 in state `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

Our environment

Our flatland environment is based on the base RailEnv class and there we wrapped the main functionalities and eased the interaction from the other classes. In order to parametrize the environment's details as much as possible we also used a yaml file *"env_parameters.yml"*.
The main functionalities are:

- **init**: in this method the parameters of the environment are passed to initialize the various controllers. We have implemented some environmental controller to flexibly add and/or remove various features in the train/evaluation phase, preserving the original Flatland's implementation. Since the controllers are tied to the solving approach they are passed from the main as parameters and instantiated inside the Flatland RailEnv class. However our implementation of the two proposed approaches share the same controller since the same features are observed and the main configuration can be tuned just acting on the parameters inside the .yml files. Our controller:

  - **NormalizerController**: this controller serves to normalize the observation, there are two implementations, one for the tree and one for the graph. The implementation is chosen through a parameter specified in the method to call. This implementation is more helpfull because it is very modular and

indipendent of the other configuration. It will be explained in detail in the chapter 6.

  - **DeadlockController**: this controller serves to individuate the deadlock state of agent and report them to the enviroment. There are two implementations, one general and one for graph observation. It will be explained in detail in the 3.5.
  - **StatisticsController**: this controller is used to compute the metrics that will be passed to wandb in order to analyze various runs and evaluate our solution implementation. It will be explained in detail in the 3.2 and 3.3.

- **create_rail_env**: in this method, the map, predictor and observation are initializated through parameters read and set in the main class. The map is created randomly but having characteristics to follow that are set in *env_parameters*. After this step all variables are passed to a costructor method for our enviroment, the parameter that we use to configure the enviroment are described in 3.1.

- **step**: in this method, there were calculated observations, standard rewards, standard info and the list of agents which have finished their run, through super method of the environment. Then, our custom actions are performed on the dictonaries before the return of the step results. Additional information are extracted from the observations and encoded in the information dictionary with method *extract_info* since they will be necessary for the following steps. The deadlock situation is updated with the aid of the deadlock controller, and rewards and statistics are computed properly. At the end a normalization procedure for the observations is applied in order to feed the training model.

- **extract info**: it takes as parameters the information and observation, which are used to fill the following variables included in the dictionary as additional information:

- *decision_required*: this variable is used to determine if agent is into a switch so the observation is not None. This flag for every agent, if is true it allows to call the act of the policy.
- *shortest_path*: this variable is a list of minimum number of switch remain to arrive at target for every agent. It is used to calculate rewards to determine if the agent come near in at target as compared to previous step.
- *shortest_path_cost*: this variable is a list of minimum distance remain to arrive at target for every agent.
- *shortest_path_pre*: this variable is a list of minimum number of remaining switch to arrive at target for every agent in previous step. It is used to calculate rewards to determine if the agent come near in at target as compared to current step.
- *shortest_path_pre_cost*: this variable is a list of minimum remaining distance to arrive at target for every agent in previous step.

- **reset**: in this method, the info and observation are reset at start state through super method of the environment of flatland. Consequentially the external controllers for deadlock and statistics management are reset as well. This method is called at the end of every episode.

- **render env**: in this method through a parameter, the GUI is launched.

3.1 Parameters

There is a file `env_parameters.yml` that contains all configurations of our environments. In details it contains three type of environments defined by us for testing purpose:

1. Small size:

- n_agents : 3
- $width$: 40
- $height$: 40
- n_cities : 2
- $max\_rails\_between\_cities$: 2
- $max\_rails\_in\_city$: 3
- $variable\_speed$: $False$
- $malfunctions\_enabled$: $True$
- $malfunction\_rate$: 0.005
- $min\_duration$: 20
- $max\_duration$: 50
- max_state_size : 24

2. Medium size:

    - n_agents : 7
    - $width$: 60
    - $height$: 60
    - n_cities : 5
    - $max\_rails\_between\_cities$: 2
    - $max\_rails\_in\_city$: 3
    - $variable\_speed$: $False$
    - $malfunctions\_enabled$: $True$
    - $malfunction\_rate$: 0.005
    - $min\_duration$: 15
    - $max\_duration$: 50
    - max_state_size : 48

3. Big size:

    - n_agents : 10
    - $width$: 80
    - $height$: 80
    - n_cities : 9
    - $max\_rails\_between\_cities$: 5
    - $max\_rails\_in\_city$: 5
    - $variable\_speed$: $False$
    - $malfunctions\_enabled$: $True$
    - $malfunction\_rate$: 0.0125
    - $min\_duration$: 20
    - $max\_duration$: 50
    - max_state_size : 72

In addition there are a some parameters to calculate the rewards:

1. Rewards:

    - $deadlock\_penalty$: -10
    - $starvation\_penalty$: -0.5
    - $goal\_reward$: 10
    - $reduce\_distance\_penalty$: 0.5

3.2 Metrics

The metrics are fundamental to understand how enviroment and policy are behaving so we implemented a *StatisticsController* to compute and print the metrics and evaluate the algorithm's performance:

- **normalized_score**: is the sum of the rewards accumulated by all agents during the episode divided by the worst score obtainable, computed as the product between the number of agents and the maximum number of steps in the episode. In the worst case, all agents do not reach their destination, therefore for each step they get a negative reward.

$$\frac{score}{max\_steps \cdot n\_agents} \tag{3.1}$$

- **accumulated_normalized_score**: is the mean of **normalized_score** obtained up to that point.

$$\frac{\sum normalized\_score}{N} \tag{3.2}$$

- **completion_percentage**: is the percentage of agents who reached their destination in the episode.

$$100 \cdot \frac{tasks\_finished}{n\_agents} \tag{3.3}$$

- **accumulated_completion**: is the mean of **completion_percentage** obtained up to that point.

$$\frac{\sum completion\_percentage}{N} \tag{3.4}$$

- **deadlocks_percentage**: is the percentage of deadlocks that occurred in the episode.

$$100 \cdot \frac{n\_deadlocks}{n\_agents} \tag{3.5}$$

- **accumulated_deadlocks**: is the mean of **deadlocks_percentage** obtained up to that point.

$$\frac{\sum deadlocks\_percentage}{N} \tag{3.6}$$

The *StatisticsController* also computes the probability distribution of the actions taken during each episode. We integrated our solution with **TensorBoard** in order to be able to analyze the evolution of both training data (losses, expected values, memory sizes, exploration rates ...) and performance metrics.

3.3 Evaluation and platforms

To run the experiments we did have at our disposal our personal computers which are powerful enough to not represent a limitation for smaller environments therefore we didn't face a lot of difficulties for the initial debug of the code and the first test for the decision of the metrics. In bigger instances primary focused to the evaluation of the models we instead experienced a slowdown in the running times. We observed that the major bottlenecks are in the Flatland code for which it is necessary to have more CPU power.
Due to this reasons we decided to limit the complexity of the experiments especially in terms of number of agent and map size, considering just the following environments with test sessions of 1000 iterations each.

1. Medium size: 60x60

2. Big size: 80x80

(detailed features of each one can be found in 3.1).
In order to evaluate different algorithms, combinations of hyperparameters and strategies we looked for a tool able to store, track, share and effectively compare different runs without the worry of continuously make notes on external files of our progresses. We found such a tool in Weights & Biases. Subscribing to a free account provides an effective and very intuitive way of monitoring a Deep Learning project.
Weights & Biases is used by OpenAI and other leading companies since it's wide range of supported platforms. Indeed we were able to integrate it rapidly in the project, connecting it to the TensorBoard logging and immediately start testing.

Additionally to a rich customizable interface to plot graphs it also provides hyperparameter tuning, called Sweep.

3.4 Action

The Flatland environment provides for each agent five different actions (already described in detail in 2.3).
In the implementation phase we reasoned about the utility of each one in order to possibly reduce the action space that our RL model has to learn. The `DO_NOTHING` is not necessary to reach a solution, because the behaviour of an agent can easily manipulated changing his movement status from the action `STOP_MOVING` or `MOVE_FORWARD`. Hence being useless and possibly damaging the overall performance we decided to consider its removal. Thinking further, the `MOVE_LEFT` and `MOVE_RIGHT` actions can be thought as ambiguous command actions where they are forbidden, since the environment will automatically mask them as forward commands. Therefore 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. That, has been observed as a very common phenomenon due to the presence of long straight paths where the agent is allowed only to stop or move forward. Anyway even the stop action in the middle of the rail does not have much sense since the unique good reason to perform a stop voluntary would be the one of give the precedence, and that can happen only in proximity of a switch cell.
For this reason we considered the possibility to force agents to only decide and learn before and over 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 a strategy of action masking to avoid illegal actions. An alternative, very common, strategy to address invalid actions would be the one of 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 resulting in a much more longer training time and the possibility that the agent converges to a wrong policy. In-

valid action masking, instead, helps to avoid sampling invalid actions by "masking out" the network outcomes corresponding to the invalid actions.
Following our reasoning we evaluate the action only before or on the switch and in these case there is a flag *decision_required* equals True.

- **"decision_required" = True**: the action choice is up to the policy on the basis of the observation provided.
- **"decision_required" = False**: the action is decided by the environment. Assigning the constant `RailEnvActions.MOVE_FORWARD` the agent will continue on the road following the direction of the road.

3.5 Deadlock Controller

In Flatland, deadlocks are a truly catastrophic event because the agents involved can no longer move and can represent an obstacle for the others during the rest of the episode. Deadlocks detection is an additional riddle in the Flatland for which there is no standard algorithm. We have tried to implement a detection system as efficient as possible able to identify failures and conflict states with the minimum overhead. That was possible especially with the aid of a custom observation. Overall, we have implented two type of detection systems based on the type of observation used.

- **DeadlocksGraphController**: this controller is used in combination of a graph structured observation. It make use of the observations of each agent controlling the presence of labels in the in the graph nodes, for instance the observation of an agent in a deadlock situation will contain a node marked with the `DEADLOCK` label.
  Remarking that for this type of observation each node represent a switch in the rail, we decided to mark a node as deadlock when an agent taking a road, with his direction of movement, will surely face another agent in opposite direction. This type of observation

however is irremediable and can develop only in a full deadlock status when both agents in the road are one next to the other. That data can be found in the same labeled node under the name of *steps_to_deadlock*. When that value is 0, coherently, the list maintained to register the deadlock status of the agents is updated, and these agents won't be considered anymore in the next steps.
To prevent such situation the model is required to learn with the aid of conflict nodes in the observation. Such label (`CONFLICT`) means that there is a possibility to have a deadlock situation for the current agent taking the direction of that switch because there is at least another agent facing that switch.
The `STARVATION` label assignment suggest instead a situation in which an agent is not able to arrive to his target position. That is possible for example in cases where two agents or more are in deadlock and block the unique road available to reach a station. When such label is present in the observation the agent situation can only develop to a deadlock status but the RL model shouldn't be penalized for that because no other option would be available. In such case we decided also to change the target to the nearest deadlock position with the objective to not cause more damages to other agents. This implementation of the deadlock controller allow also to interrupt an episode early when all the agents are in `DONE` status or `DEADLOCK` or `STARVATION`.

- **DeadlocksController**: this controller is the default used with any type of observation. It use the distance matrix for determinate if an agent is in deadlock, checking adjacent cell and comparing directions of other agent. In case of a deadlock, the corresponding flag of the agents involved is marked true. In this implementation hasn't been taken care of the conflicts or starvation situations, since they will in any case develop in a deadlock or being interrupted by the end of episode. This obviously involves an higher risk to incur in deadlock cases or slow down other agents and in a deterioration of network performance but is more simple.

3.6 Rewards

As mentioned before 2.4 the Flatland environment provides a basic rewards system briefly describable with the following points:

- Every step agent receives a negative reward proportionate to his speed if he has not reached his destination.
- Each agent receives a reward equal to 0 if he has reached his destination.
- If all agents have reached their destination they receive a reward equal to 1.

We think that this reward system is lacking of some important details to fully represent the complexity of the problem because there is not much distinction between the states in which the agents may be while navigating in the environment.
Intuitively an agent have to learn two behaviors, not necessarily in this order:

- Reach his destination in the shortest time possible.
- Avoid collisions with other agents.

Let's consider a small environment with 3 agents, in this case the main behavior is the first because the probability of a collision is not very relevant, but if we consider the same environment with 10 agents the skill to avoid deadlocks is decisive for the overall performance.
According to the Flatland's rewards system there is not difference between being deadlocked and navigating the map without reaching destination from an agent's point of view in terms of rewards.
In order to stimulate the learning of the desired behaviors we have tried to modify the Flatland's rewards system by using a method known in literature with the name of **Reward Shaping**. Crafting rewards is not easy because as a consequence we could get trapped in the "Cobra Effect":

> *"Historically, the government tried to incentivize people to assist them in ridding the area of cobras. If citizens brought in a venomous snake they had killed, the government would give you some money. Naturally, people started breeding venomous snakes.*[1]

Indeed, this method may cause an undesirable effect: stimulating the learning of one behavior can cause the learning of another wrong.
Taking all of that in consideration, in our implementation we added a method in the env class called **compute_rewards** to flexibly choose during the training phase whether to use the rewards shaped or the standard Flatland's reward system.
In the first case it is possible to choose how to shape the rewards by setting the following parameters:

- *deadlock_penalty*: to penalize agents in deadlocks by value of variable.
- *starvation_penalty*: to penalize agents in starvation by value of variable.
- *reduce_distance_penalty*: to reward agents who are moving towards their target by multiplying the value assigned to the reward associated with the agent calculated in the step.
- *goal_reward*: to reward agents who have arrived at their destination by assigning them a positive reward.

[1] https://medium.com/@BonsaiAI/deep-reinforcement-learning-models-tips-tricks-for-writing-reward-functions-a84fe525e8e0

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.

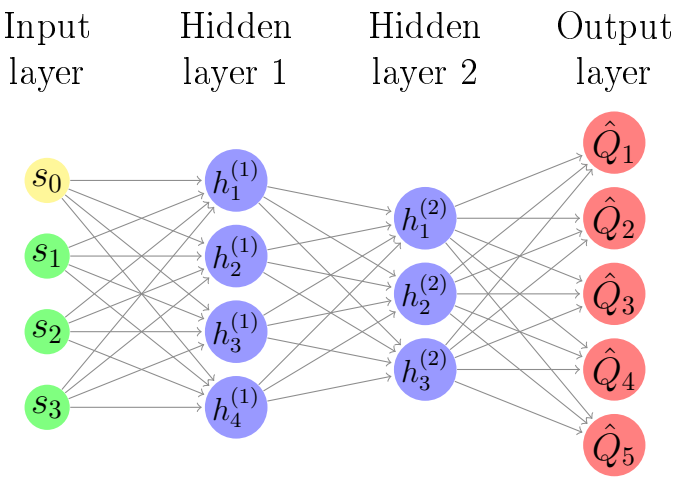
4.1 DQN

[1] described 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, that is the difference between the current prediction and

the expected optimal target. 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 128 and 64, 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* class implements the neural network, using **Huber loss** and **Kaiming initialization**. The former is, as explained in [2], a loss function that 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.

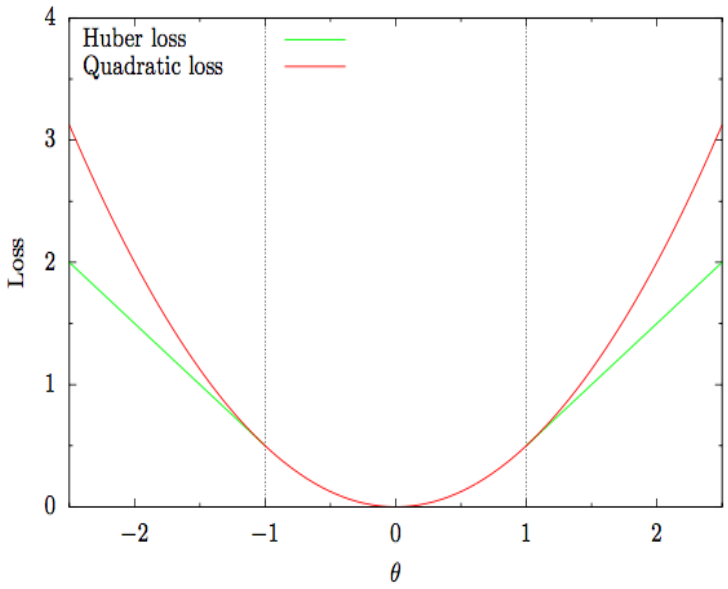


Figure 4.1: Huber and squared loss

Kaiming initialization is instead 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. It does so by doubling the variance, since ReLU halves it w.r.t. to the original standard normal distribution (see 4.2).

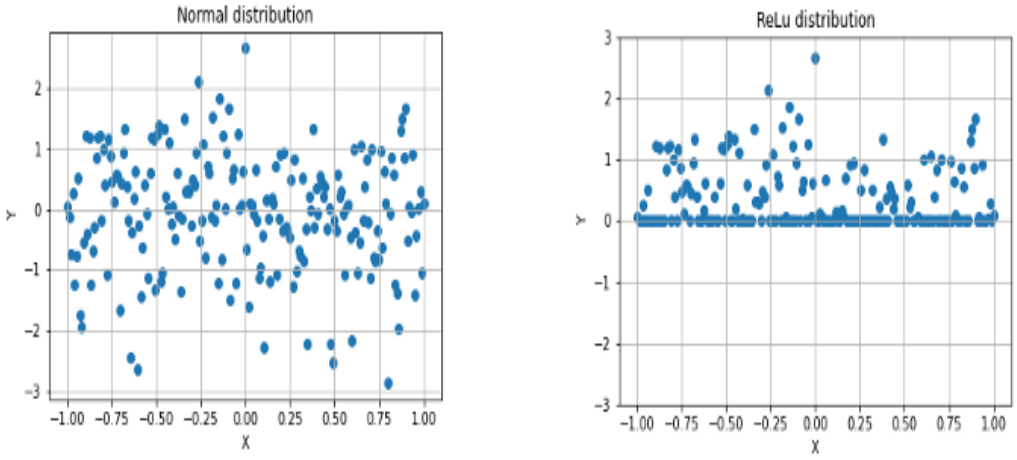


Figure 4.2: Normal and ReLU distributions

The *DQNPolicy* class implements the main methods needed for the training loop:

- **act**: implements the **ϵ-greedy** action selection policy, by generating a random number, and then comparing it with ϵ. This hyperparameter is then iteratively decreased at the end of each episode in order to incrementally rise the probability of allowing the model itself to pick the action.

- **step**: performs the learning step, though only once every *update_rate* time steps, a hyperparameter initialized in the yml configuration file. Before that, if prioritized experience replay is used, then the priority is computed and the current experience tuple stored in memory with that priority.

- **learn**: called by the step method, it actually performs the learning step with an experience sampled by the memory. It computes the temporal difference target and then performs a gradient de-

scent step. It also updates the memory with the newly computed temporal difference, which serves as a priority for the experiences.

We also used a technique called **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.

4.1.1 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; instead, greedy prioritization risks overfitting since the same higher priority experiences are always replayed.
Concretely, in stochastic prioritization, the probability to sample transition i is

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

with $p_i = temporal\ difference\ + \epsilon$.
α is a hyperparameter that controls the tradeoff between random ($\alpha = 0$) and priority choice ($\alpha = 1$), whileϵ is a small positive hyperparameter to always keep a non-zero priority, thus allowing to revisit even the lowest priority experiences: this specific variant is called **proportional stochastic prioritization**. 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, that favors higher priority samples. 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.
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.

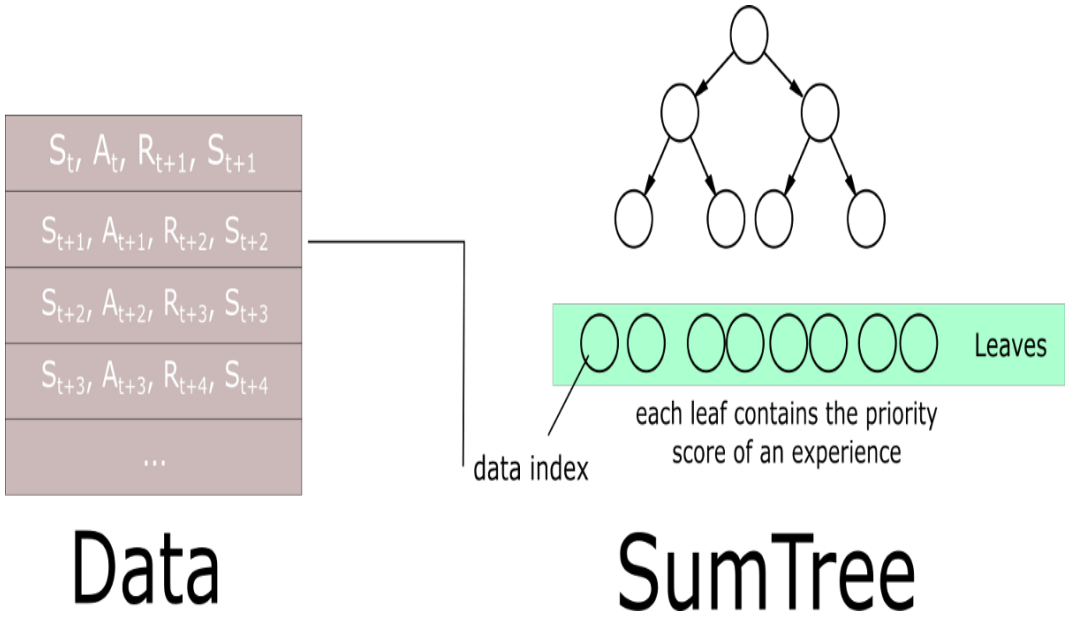


Figure 4.3: Sum Tree

The *SumTree* class stores 2 numpy arrays, one for the experiences (so just the leaves), and the other represents the tree and thus stores the priorities. Inside the class there is a pointer to the current leaf that is constanly updated and used whenever a new experience is added to the

replay memory. If the number of samples exceeds the capacity of the memory, the last experience is overwritten by the new one.
PrioritizedReplay is the class implementing the probabilistic sampling of the experiences from the memory (sumtree): the total priority is divided in $batch\_size$ segments, so that for each element in the batch we have a segment from which we can sample uniformly a priority, which is in turn used to pick a leaf from the tree.

4.1.2 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.

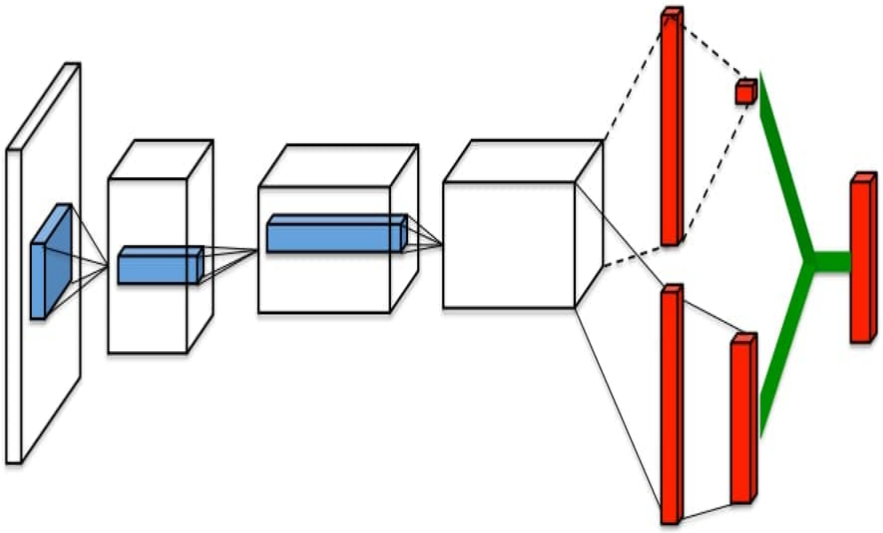


Figure 4.4: Dueling DQN

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 Graph observer

One of the main objectives of the Flatland-Challenge is to find a suitable observation (relevant features for the problem at hand) to solve the task. For this reason the Flatland library provide full flexibility when it comes to building your custom observations. Whenever an environment needs to compute new observations for each agent, it queries an object derived from the `ObservationBuilder` base class, which takes the current state of the environment and returns the desired observation.
We had at our disposal for testing the environment many observer:

- TreeObsForRailEnv
- GlobalObsForRailEnv
- LocalObsForRailEnv

Anyway for our comprehension of the environment these default version could not be able to perform at best with a RL approach since they compute a view of the environment focused on unnecessary details and more importantly they describe just the agent neigbourhood without giving contextual information to the external *entity* which have to choose the action to perform. Some of them were considered for a deeper analysis and testing but as we have hipotesized the performance were not good.

We opted for a custom observation based on a graph inspired by the TreeObserver but with a totally different structure. In particular what we requested from the observer entity was:

- ease the search path as possible since the management of transitions and direction feasibility can be quite a complex job in bigger environments.
- be able to retrieve the recurrent information for a certain point in the space in linear or at maximum in polynomial time.
- reduce the search space and decrease the search time for the identification process of the shortest path at each step of the agents, but still maintaining a good level of explainability in the resulting view.
- build an observation with data and structures properly formatted with the aim of feeding a network model with it and later on reduce the effort necessary for the training.

Our implementation can be found under the class `DagObserver`. It extends the base class ObservationBuilder and implement the standard behaviour including the initialization of the environment variables and data structures and the it's reset procudure to be executed at the end of each episode. Moreover there are the *get* methods (single and multi agent) which are invoked at each environmental step to obtain the observation of each agent.
In our implementation the reset method is also in charge of the creation of the initial *general_graph* which will be used during the entire episode to calculate the paths and foresee the movements of the agents. Furthermore has been dedicated particular attention to the the order of computation of the single observations: the *get_many* method instead of iterating over the cardinal dictionary of agents and call the *get* method, use a list of agents ranked by priority. Our intuition was inspired by common rules used while driving the car:

- a slow driver self aware of it's ability give the priority to the others when in doubt

- a normal driver knows that there in presence of some signals like the sirens of an ambulance/police he should absolutely give way
- a fast driver is probably more careless/irresponsible. He will surely arrive faster at the destination assuming he won't make an incident...

In the same manner we calculated an index which summarize the characteristics of an agent and influence the level of attention that he use to avoid conflicts with others. The more priority has an agent the less he cares about possible conflicts. Although this could seem an egoistic implementation, it must be considered that the other agents, being more careful (less priority) will be able to detect and avoid the incident giving way to the opponent.
Then, after being computed, the observations are stored in the *prev_observations* dictionary for future uses and returned as output value to the caller, the `FlatlandRailEnv`.

5.1.1 General graph

This graph is an abstract but detailed view of the rail grid environment. It's built with the specific aim of store and rapidly retrieve information from whatever initial position or path taken on consideration for the exploration. That is of primary importance since the structure of the graph itself is a simplification of the real transition system provided by the environment and would be used during each step search process. That indeed require to have an optimal time of access for certain information being accessed frequently. Some of them are also encoded in the structure of the graph for instance the orientations and access point associated to the nodes which dictate the possibilities of movement of an agent during a traverse process.

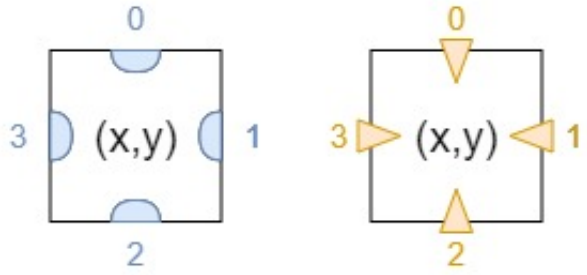


Figure 5.1: Rail orientations

Also for each pair of switches directly linked in the rail there is a connection in the graph too.
The main point in favour of such structure is that it results in a reduced space of the real environment and it can be used with ease for the process of path discovery from source to target. De-structuring the rail environment we can indeed extract possible valuable paths which will be verified in a second moment controlling the feasibility of each transition from node to node.
A failure is encountered when, for instance, the path use two consecutive switches between which doesn't exist a feasible transition considering the orientation of the agent. Note that, that is not properly a bad event since the observation algorithm is able to explore the graph resuming from the failure node reached, and can then calculate a new shortest path to the target. Moreover the invalid transition is saved and will be taken in consideration during the future path discovery processes.
On the graph are also registered events of deadlocks or starvation situations: updating the status of the environment those events can be handled properly minimizing the probability of future problems derived.
At each step of an episode that process is replicated for each agent in a way to improve the structure reliability of the *general_graph* and at the same time the current observation from the agent position is built. Anyway most of the changes to the initial graph are not permanent since it will be resetted at the end of each agent step allowing different paths discovery in the successive situations. That, has been evaluated as a good compromise between state management retainment and the

computational cost of evaluate each time the environment, due to it's dynamicity.

Deadlocks and starvations

The general graph allow also to spot edge cases in which the target is unreachable:

- Deadlock: detected when from the start point of the agent another agent incoming in the opposite direction is found before a switch cell.
- Starvation: happen when the procedure of minimum path search is unable to return a value

In case of a deadlock the agents are forced to move until they are stuck one in front of the other with no more possibility of action. Such will is motivated by the handling of the starving agents. We thought that in cases which the target reachability is negated for an agent but he still can move, his presence can be harmful to other agents therefore he will be marked as "starving" and sent to "die" in a deadlock edge position. Compacting the agents in with deadlock status into an edge allow to have more space for other agents in such dead rail.
The only situation that will cause a permanent resize of the graph is a deadlock event with the parameters *steps_to_deadlock* to 0 and *first_time_detection* True. That happen when the agent is stuck with another agent (no possibility of going forward) and the decision of enter in such rail, with the consequent deadlock detection, has just been made (hence the previous cell is a switch). To handle that situation the edge in the same direction as the agent is removed and recursively the transitions of the nodes involved are controlled, eventually deleting also the descendant edges if such node hasn't anymore a valid transition in the specific orientation.
The deadlock event can also happen on a switch: in such case all the rails going toward that switch are removed.

Structure

The structure of the graph can be seen from the following image showing the rail and the corresponding graph representation:

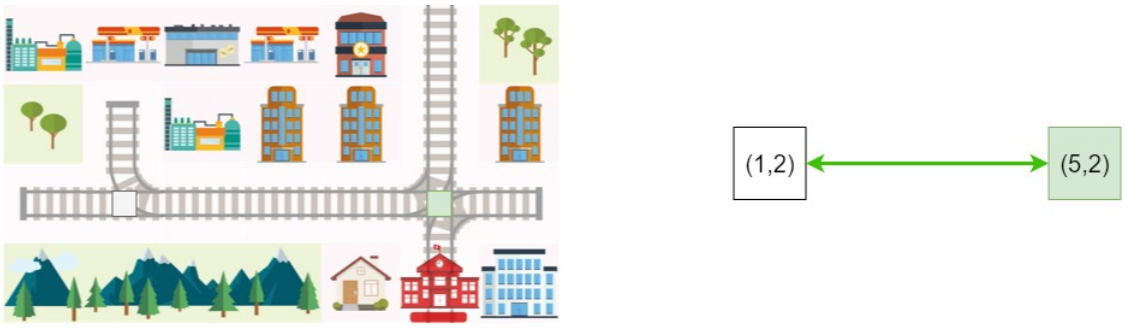


Figure 5.2: General graph

For each switch there is a node with associated a pair of coordinates (x,y) while each edge represent a rail between the switches. It can be observed that two switches could be connected by more than one rail dependently to the access point used for the transition. To handle that requirement, for the graph data structure has been used a `MultiDiGraph`. The nodes contains information relative to:

- Transitions: a matrix 4 by 4 of zeros or ones.
- Node transitions: a matrix 4 by 4 where the values are the nodes reachable with a transitions from a certain orientation and access point.
- Dead end: the presence of dead ends which can be used to return to the same switch, specified with access point to use and cost associated.
- Targets: the presence of targets in a certain direction from the current node.

The edges contains information about:

- Weight: the steps between the two nodes.

- Access point: specify which access point will be used for both ends of the edge. It's useful since the exit direction wrt the edge direction is also the key identifier of the edge itself.

5.1.2 Direct graph graph

The expressivity of that graph is much higher but it's existence is retained to the observation of the single agent in the current environment situation. It lack of all the navigation information that can instead be found in the general graph, since its main focus is to represent explicitly a path from a start point to the target driving the decision of the model for the next action selection for the agent. The construction process aim to find the best (shortest) path from the agent position to one of the possible switch positions that give access to the target, furthermore has been implemented a behaviour aimed to explore other possible paths and also possible interactions with other agents are taken in consideration.
In detail the first search of a path is executed scanning the initial graph without the other agents, then a second search is made removing the edges in use by the other agents but in the opposite direction, in a way to find a path which avoid them. The final step is about marking the nodes with the *conflict* label if they appear at a certain distance from the start node both in the current agent observation and in the observation of another agent with higher priority. The idea as explained before is that who has a lower priority should take care of possible conflicts with who has an higher priority.

Structure

The structure of the graph can be seen from the following image showing the translation from the the general graph for the agent represented:

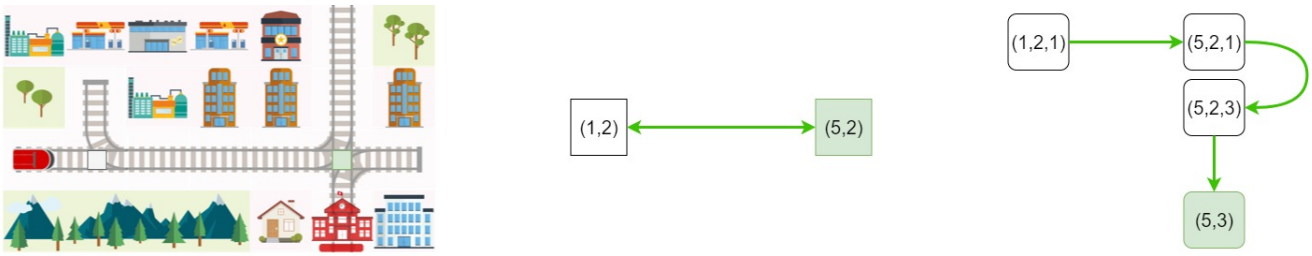


Figure 5.3: Direct graph

Each node is represented with a triple of coordinates and orientation (x,y,dir), therefore each switch is represented uniquely considering the direction of access. The graph/rail correspondence is valid for all the nodes except for the target position which is represented with a tuple of coordinates being not a switch cell. The starting node is instead approximated to the nearest switch which involve a decision for the agent.
Note that given the construction procedure used, the graph can also be commonly considered as acyclic.

The following are the labels and the piece of information stored in the graph. Their primary role is to enable statistics computation and allow the identification of the precise role of a node in the graph, giving a reference structure for the model learning process.

- `START`: represent the start node and contains the information of the current agent like "velocity", "switch_distance", "nr_malfunctions", "next_malfunctions", "malfunction_rate", "malfunction".
- `TARGET`: assigned to the target node.
- `DEADLOCK`: it indicates that the agent is in an unrecoverable situation. Additional info provided are "steps_to_deadlock" and "first_time_detection".
- `CONFLICT`: the choice of that node can give rise to a deadlock situation with another agent. The following information describe the characteristics of the opponent: "velocity", "conflict_distance",

"target_distance", "nr_malfunctions", "next_malfunctions", "malfunction_rate", "malfunction".

- `STARVATION`: that label can be found in the start node in case it doesn't exist a path for the agent to reach it's target.
- `DEAD_END`: nodes that can be reached with the exploit of a dead end in the previous step.

5.1.3 *get* procedure

The `get` procedure is the main call of the observer and is in charge of compute and return the observation for an agent. The method can be summarized in the following points:

- The directed graph is created and the general graph structure and data are copied in a new object so the subsequent modifications won't affect the original structure.
- From the position of the agent the path is scanned until a switch cell which need a decision from the agent. In that rail portion possible combinations of events are handled and the first switch position is then added as starting node. In case of a deadlock the start node is labelled properly and the process return the graph as it is, instead when target is encountered or the number of steps performed to reach the switch is greater than 1 a None object is returned since no action is required.
- In the remaining cases (before a switch or over it) the target is computed and added to the graph already linked to the *ending_points*: the switches from which is possible to reach the target position or the deadlock positions in case of a starving agent.
- The first step of path search is forwarded on the general graph without the other agents. In the method *_build_paths_in_directed_graph* the search process extract a sequence of nodes as a valid shortest

path and that is then checked evaluating the validity of the transitions between the nodes. This process proceed in parallel with the creation of the direct graph which is compiled with only the decision switches found in the path. In case of a rejection of a sequence of nodes for missing or invalid transition the two nodes and current orientation are saved and that denied rail will be considered in the next search iterations. The path followed till the negated node is maintained as valid, and a new iteration is forwarded with ever the same targets (the *ending_points*) but with the node found to be unreachable as start position: if the path in which it was included was opitimal, but invalid, probably there could be a slight deviation which make it valid and also optimal in cost value.
The search is repeated until all the nodes of an extracted path are in the directed graph and they are all connected sequentially.
Note that in case of a failure in the path detection process, the observation graph is labelled with the starvation flag. The *ending_points* are replaced with deadlock positions and the search is reiterated.

- The second step is forwarded as previously but this time the general graph has been updated removing all the edges which will take to face another active agent. The edge removing process is iterative and therefore all the resulting unusable nodes are removed.

- The third and last step is in charge to label the conflict nodes. There we make use of the previous cited ranking by priority of the agents. Since the computation of the observations happen by priority order, we have at our disposal all the observations of agents with higher priority. Consequently we can match the nodes in the graphs, just by x and y, to find common nodes and therefore possible conflicts in a certain range from the start node.

- The resulting observation can then be returned to the train cycle

Can be observed that this implementation separate the actual search of the minimum path from the detection of a feasible way. Extracting a minimal path each time and try to traverse the nodes with that trace allow us to discover other sub-optimal paths, give more way to learn to the model, and indirectly allowing the agent to reach his target more easily and with optimal results.

5.2 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 adjacent 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.

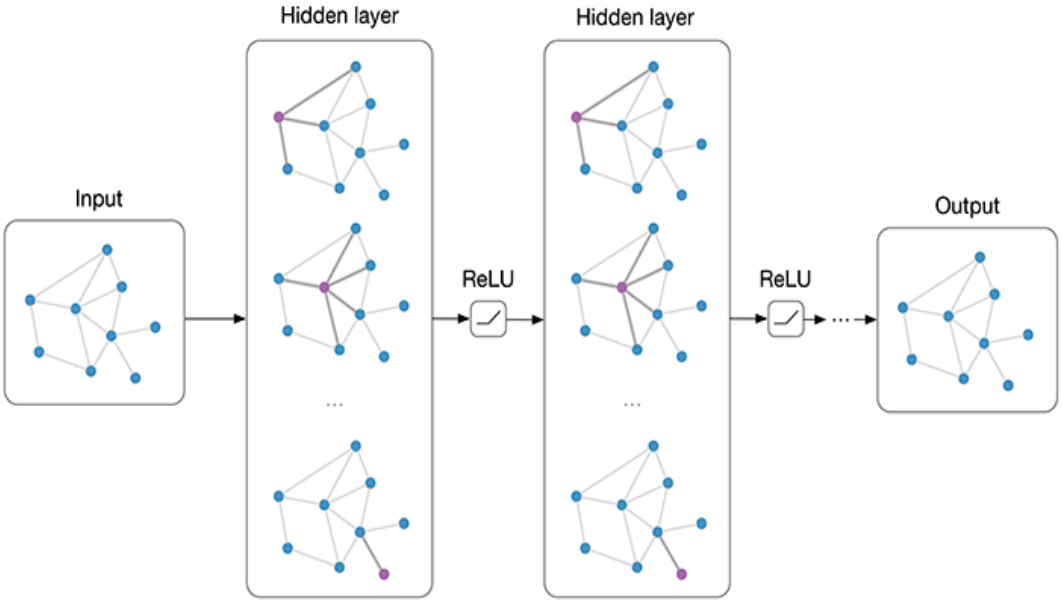


Figure 5.4: GCN

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.

Chapter 6

Normalization

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