Università degli Studi di Bologna

Computer Science and Engineering Department

Artificial Intelligence

Relation flatland challenge

Artificial intelligence

Supervisor
Prof.Andrea Asperti

Students
Paolo Di Dio, Luciano Massaccesi, Davide Angelani

Academic year 2020-2021

Abstract

Flatland challenge is a competition created by AIcrowd in cooperation with the Swiss Federal Railway Company (SBB). The aim of the competition is to address the vehicle rescheduling problem. Anyway solving the problem while addressing all aspects of the real world would result in dealing with extremely complex setting. In order to develop new algorithms faster, a simplified 2d world which simulates 2d multi-agents railway is proposed. To fasten the development of new solutions, a first benchmark is made available.

Indice

Elenco delle figure

Capitolo 1

Flatland environment

In this chapter the flatland environment is presented

1.1 The world

Flatland is a 2D world where agents, in this case trains, have to navigate the map towards their target. The agents will learn the best policy to travel with a reinforcement learning algorithm.

This 2D world is composed by many different kind of cells, but we can distinct them in two categories:

* Non-rail cells: it includes trees, cities and all the cells that train cannot cross
* Rail cells: cells that permit the trains to pass over them.

In addition, every rail cell can be occupied by targets and trains.

1.1.1 Rail cells

Every rail cell is bind by a transition matrix which describe, given the arrival direction to the rail cell, to which direction a train can move. Depending on the transition matrix, rails can encode curves, straight rails, dead ends or, when there are more than 2 arrival directions, switches. Switches are the most relevant cells because train can make decision there.

1.1.2 Trains

Trains start from an initial position and must reach their target while avoiding other trains. Every train at every time step can choose one of many possible options:

* move forward: turn back in case of dead end
* move left
* move right
* stop: stops moving, always possible.
* no-op: keep doing what it was doing before, always possible

Collisions with other trains are automatically avoided by stopping the train, anyway the train could block each other and enter into an irreversible state of deadlock. In addition, trains can have malfunction which cause the train to stop for a certain amount of time.

Capitolo 2

Benchmark

In order to fasten the development of new algorithms, a benchmark solution is offered and described in this chapter. The proposed solution uses a reinforcement learning algorithm. As such, there are two main aspect that are handled and can be improved: the observation of the environment which is needed by the agents to see the world and a way of learning from the experiences, based on a system of rewards.

2.1 Reinforcement learning

Reinforcement learning problems involve an agent interacting with an environment which provides numeric rewards. At every step the agent gets a state or observation from the environment and must select an action according to a policy. The environment returns a reward and then enters in the new state. In Figure 2.1 can be seen a schema of reinforcement learning problems.

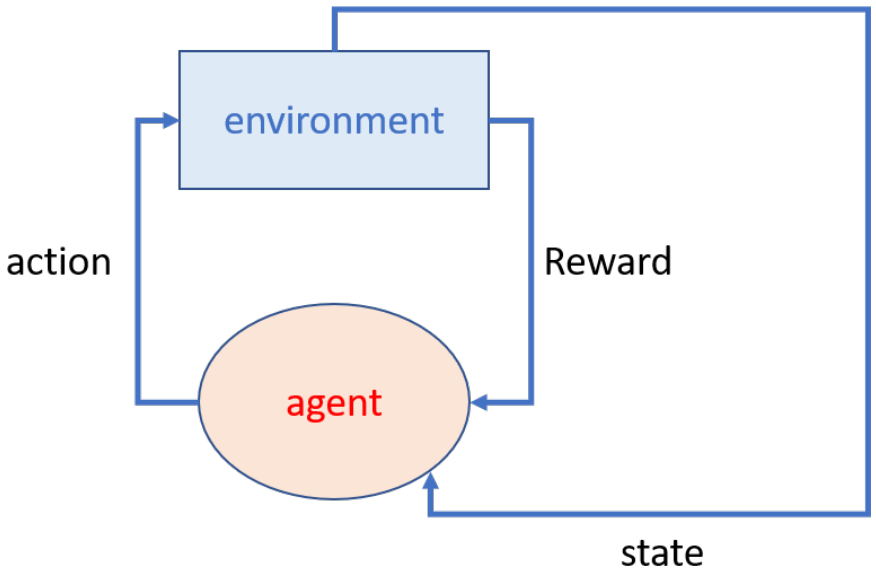

Figura 2.1: Reinforcement learning schema

2.2 Observations

The environment already offers a 2d map with all the information useful to define the complete state of the world at each moment. Anyway, getting the whole raw 2d map with all rails, other trains and targets would result in an excessive amount of information.

This brings two main problems:

* Scalability problem: when the size of the environment increases, the information that needs to be elaborated also would increase in the same way, resulting in an increase of the time needed and a worsen of results.
* Learning from raw 2d map can result difficult.

The scalability problem is solved by creating an observation tree for each agent with a limited depth where, for each node of the tree, some features are computed. The information used to create the features could be divided into local information and global information. Local information are retrieved only from the local area around the node while global information are computed from the whole environment and are not limited by the position of the node. Global information are particularly important because can give information of things outside the observation tree. In particular, most of the feature are local to give the agents a good understanding of the area close to the train. The most important global information is the minimum distance from the agent target. The idea is that the local information can be used by the agents to get a good understanding of the neighborhood in order to avoid other trains, while global information can be used to get an indication of the direction to take to reach the target.

The second problem is addressed by transforming the 2d raw map into some more elaborated features that can be used more effectively by the agents. Those features must preserve only the important information while discarding the irrelevant information. The features are then assigned to each node of the observation tree.

In practice, the observation tree is generated starting from the agent and by creating a node for each switch encountered where the agent could take different paths. Then for each possible path, a new node is created and linked to the father's node. For every node, the information of the rails between the previous node and the switch is summarized by features. The exploration ends when the maximum depth is reached or when the agent's target is found. This algorithm is implemented by exploring cell by cell starting from the agent while creating the observation tree. This process is done for every agent, for every step (every time the train moves). Anyway, this could be a redundant process as the information regarding the 2d map does not change from an agent to the other or from one step to the next one. A better implementation will be shown at chapter 5

2.3 Rewards

The reward is an important step for the learning process. A good reward function is needed to define which behaviour of the agent is good and which one is bad. In this case, the agents must reach their target while minimize the time spent to do so. A positive reward is then given when the agent reaches its target ($global\_reward$), a null

reward in case of invalid action, starting or stopping, while a negative one each step the train does not reach the target (*step_penalty*) [2]. Even if the reward system is really simple, it is enough for the trains to do what they have to do. Anyway, different reward system could be applied to force the agents to focus their learning on different things. One example will be shown in chapter 4.

2.4 Policy

Given a state of the world, the policy is a probability distribution of actions that tells which action must be taken. The best policy is the one that receive the most cumulative reward. The learning process of the algorithm consist then in improving the policy to maximize the cumulative reward. There are more ways of implementing the policy, one of the main ones is to use a neural network. In this project, the dueling double deep Q-learning network (dddqn) is used.

2.4.1 Q-learning

This algorithm is based on a table (or function) called Q(s,a) which gives for every state, and action, how good it is. The latter is defined by the expectation of cumulative reward. First, the Q table is initialized. Then, every step an action is taken and the Q table is updated as such:

$$Q(s,a) \leftarrow Q(s,a) + \alpha * (r_0 + \gamma * max_{a'} Q(s',a') - Q(s,a))$$

where α and γ are constants.
When exploring the possible states and actions, an ϵ is initialized to 1 and then decreased over time. The action is taken randomly with probability ϵ. This cause the first explorations to go completely random, then the algorithm focuses more and more on what are predicted to be good actions.
Building the Q table can require too much time when the number of possible states increases. For this reason Q can be approximated to a function, determined by a neural network.

2.4.2 DDD Q-learning

When Q is approximated by a neural network, it is called Deep Q-learning. In this case Q(s,a) is approximated by a neural network. Anyway, in simple deep Q-learning the same network is used for both select and evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. In double q-learning, two neural networks are learned by assigning experiences randomly to update one of the two. For each update one is used to determine the action, one to determine its value (needed to evaluate it) [3]. In dueling q-learning the q-function is separated into two: one is used to determine how good is the current state, the other to determine the advantage obtained by the given action [5]. The two values are then aggregated together by summing up and removing the average of the advantage. The schema of the complete network used can be seen in figure 2.2.

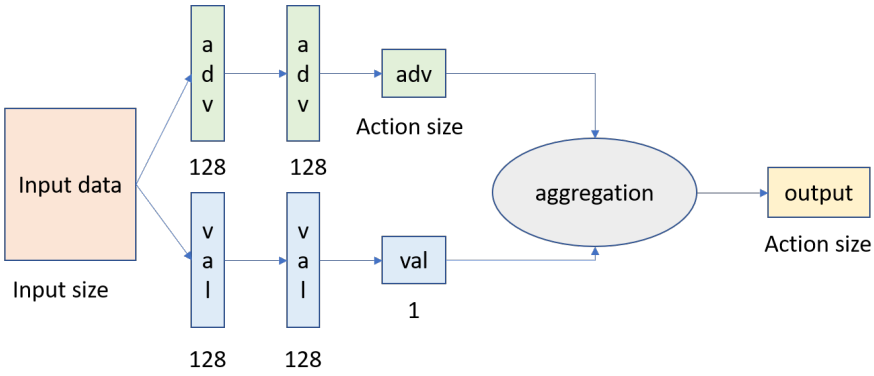

Figura 2.2: Double deep Q-learning schema

Capitolo 3

Hyperparameter tuning

In this chapter it will be shown the performance of the project as the group found, and the first attempt to improve it by running it with different parameters.

3.1 Baseline performance

First, a distinction between actual "project hyper-parameters" and "run parameters" must be done: some of the parameters that are given to run the project affects only time of computation and/or graphical output, and are irrelevant to the performance of the network.
The following are considered as "run parameters":

* use_gpu with default value of 'False', makes it possible to use GPU to run the network
* num_threads with default value of '1', number of threads PyTorch can use
* render with default value of 'False', makes it possible to render graphically 1 episode every 100

While these are considered as "project hyperparameters":

* n (or n_episodes) with default value of '2500', expresses the number of episodes to run
* t (or training_env_config) with default value of '0', expresses the environment used for the training phase
* e (or evaluation_env_config) with default value of '0'. expresses the environment used for the evaluation phase
* n_evaluation_episodes with default value of 25, expresses how many episodes are included for an evaluation of the network
* checkpoint_interval with default value of '100', expresses how many episodes per checkpoint
* eps_start, with default value of '1.0', is a network parameters that influence the maximum probability that a train has to explore the environment instead of using its own knowledge

* eps_end, with default value of '0.01', is a network parameters that influence the minimum probability that a train has to explore the environment instead of using its own knowledge
* eps_decay, with default value of '0.99', is a network parameters that influence how fast esp_start decays to eps_min
* buffer_size, with default value of 'int(1e5)', expresses how big is the memory buffer in case precedent knowledge is used
* buffer_min_size, with default of '0'
* restore_replay_buffer, with default value of '', is the name of the checkpoint file used to get previous knowledge
* save_replay_buffer, with default value of 'False', expresses if the following run must be saved in the buffer
* batch_size, with default value of '128'
* gamma, with default value of 0.99, is a network parameter known as discount factor, to propagate the rewards further in time (a discount factor of 0 makes the agent care only about immediate rewards)
* tau, with default value of '1e-3'
* learning_rate, with default value of '0.5e-4'
* hidden_size, with default value of '128', is a network parameter that represents the number of hidden layers in the network
* update_every, with default value of '8', expresses how otfen the network is updated

The baseline is considered with all "project hyperparameters" as default, while the "run parameters" are custom (for example, $num\_threads = 8$ fasten up the run).

The following 3 runs constitutes the baseline, and they will be interchangeably used to compare other runs.

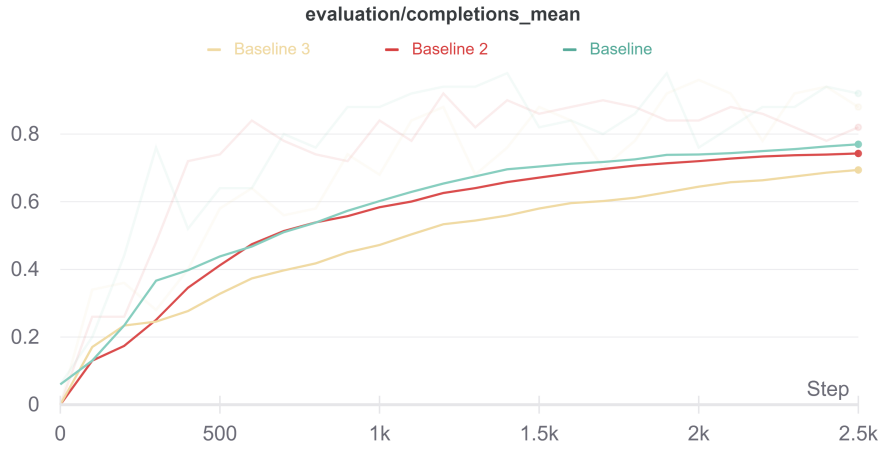

Figura 3.1: Completions on the evaluation episodes

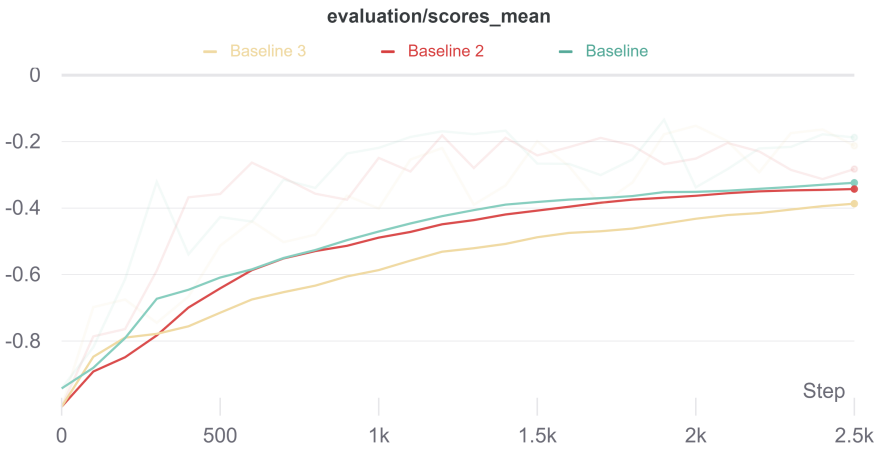

Figura 3.2: Score on the evaluation episodes

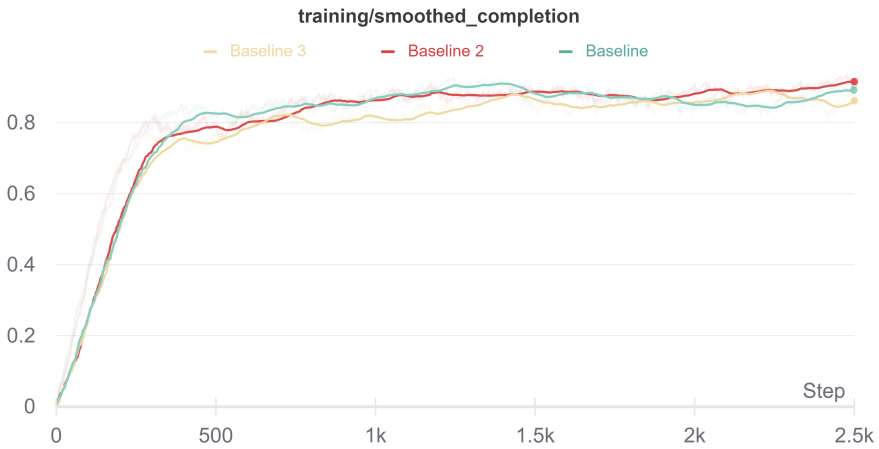

Figura 3.3: Completions on the training episodes

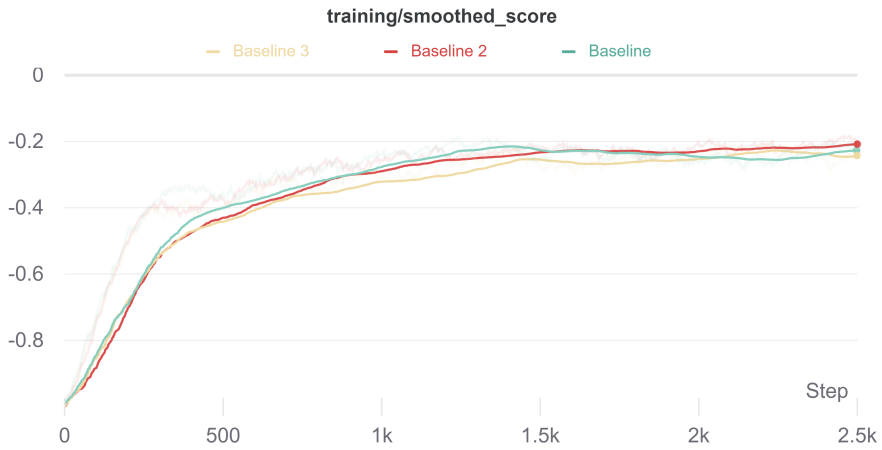

Figura 3.4: Score on the training episodes

3.2 Project hyperparameter tuning

The first question that comes to the head is: "Can we improve the baseline by changing the value of some project hyperparameters?".

Number of episodes To decide which is the suitable number of episodes to train the agents, we need to consider the improvement that a greater number of episodes can give to the model with respect to the time of computation. In this project, there has been a huge difference on the execution time in the 3 different systems we were working with:

* Paolo Di Dio: runs the baseline in approximately 140 mins (2h 20 mins)

* Davide Angelani: runs the baseline in approximately 4 hours

* Luciano Massaccesi: runs the baseline in approximately 7 hours

It is possible to run more episodes, but as shown in the next figure, the score curve becomes flat getting closer to the 2500° episode, therefore it is not interesting to have a greater number of episode. Furthermore, the map environment of the baseline is the simplest, hence the fastest, among the others used for testing.

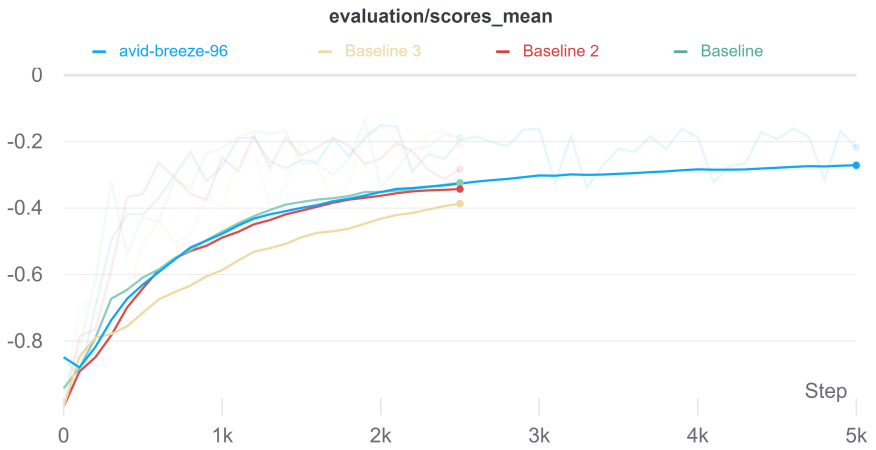

Figura 3.5: Run with 5000 episodes: the improvement is not worth the higher number of episodes

A smaller number of episodes it can be considered for the systems of Luciano, but the model doesn't reach its optimal score in that number of episodes, so, regarding the number of episodes, the standard we put for the tests is 2500.

Training and evaluation environment Those two parameters can't be tuned to reach an optimality because the variables they change are too many to decide that one environment is the best among the other. We need always to consider that the dimension of the map and the number of agents can signficantly change the aim of the training. For example, a small map with a few agents can reward more on finding the fastest path to the target, because of the smaller probability of deadlocks; but the same map with lots of agents can reinforce a "avoiding deadlock" policy. The same goes with changing the dimension of the map but keeping the same amount of agents. We can say there's a ratio of rail cells and number of agents that changes the training policy, but there's no absolute hierarchy that can discriminate a better map than a worse one, because it depends on the evaluation environment. But, if we know the environment of evaluation, or just the rail_cells/agents ratio, we can imagine which kind of map is more suitable for the training.

As shown in figure, similar environment for training and evaluation performs better than having different ones.

Environment 0 (default):

* n_agents: 5
* x_dim: 25
* y_dim: 25
* max_rails_between_cities: 2
* max_rails_in_city: 3
* malfunction_rate: 1/50

This is the environment as found in the flatland environment.

Environment 1:

* n_agents: 10
* x_dim: 30
* y_dim: 30
* max_rails_between_cities: 2
* max_rails_in_city: 3
* malfunction_rate: 1/100

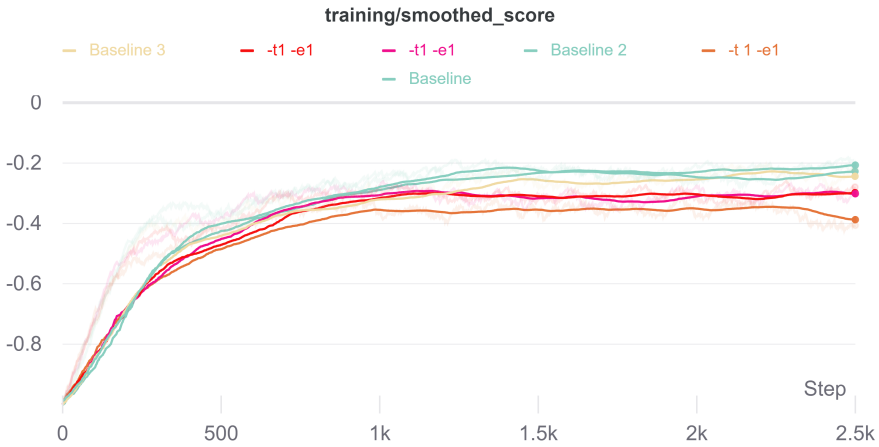

Figura 3.6: Score during training episodes with environment 1

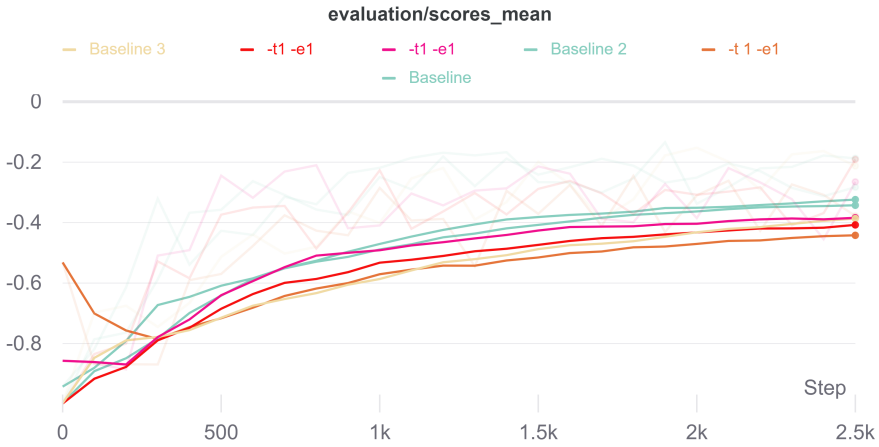

Figura 3.7: Score during evaluation episodes with environment 1

Environment 2:

* n_agents: 20
* x_dim: 30
* y_dim: 30
* max_rails_between_cities: 2
* max_rails_in_city: 3
* malfunction_rate: 1/200

This environment is modeled as an example of a heavily busy (dense) map.

Here are compared the performances of using the environment 2 only on the training or evaluation episodes, to show the impact of having a very different environment during one of the two phases.

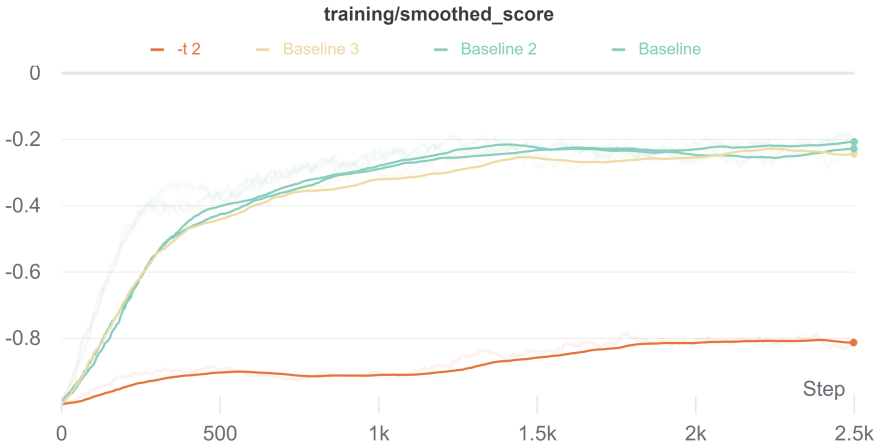

Figura 3.8: Score during training episodes with environment 2

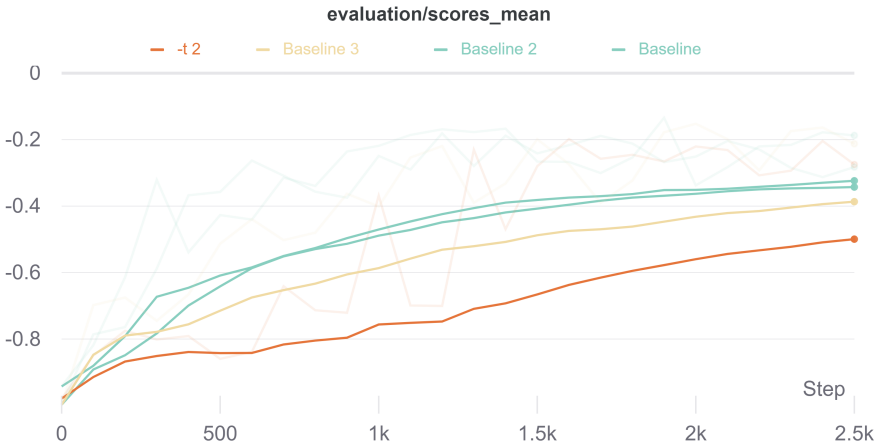

Figura 3.9: Score during evaluation episodes with default environment that has been trained with environment 2

An environment so busy as the environment 2 gives very poor results in training but if evaluated in the default environment, the loss on the performances is less. That means that it is possible to successfully train the trains in a dense map, but it requires some improvements of the model to perform as good as a sparse one.

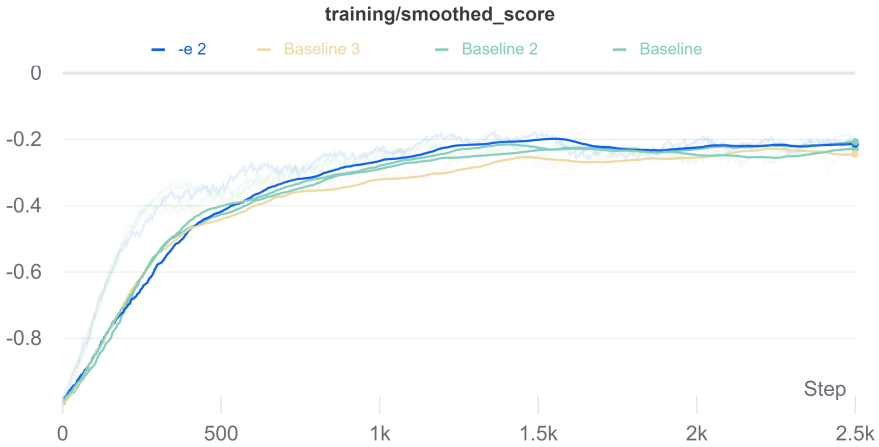

Figura 3.10: Score during training episodes with default environment

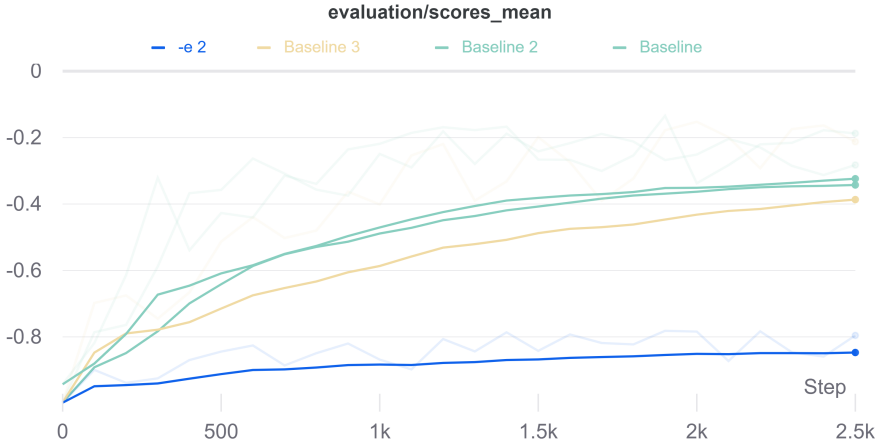

Figura 3.11: Score during evaluation episodes with environment 2 that has been trained with default environment

The training with default environment gives very poor results on the evaluating episodes using a "dense" environment as environment 2. We can say that during the training in the sparse (default) environment, training the agents how to improve their score by finding the fastest path to the target is usually the best idea. But the same agents, when they are put inside an environment that requires a policy focused on avoiding deadlocks, perform poorly.

In conclusion, we can say that a policy built on avoiding deadlocks adapts more on different maps than a policy that aims to find the fastest path to the target.

Epsilon Epsilon can be manipulated in three different ways: by *esp_start*, *esp_end* and *esp_decay*. Here are the conducted tests:

epsilon_start = 0.8

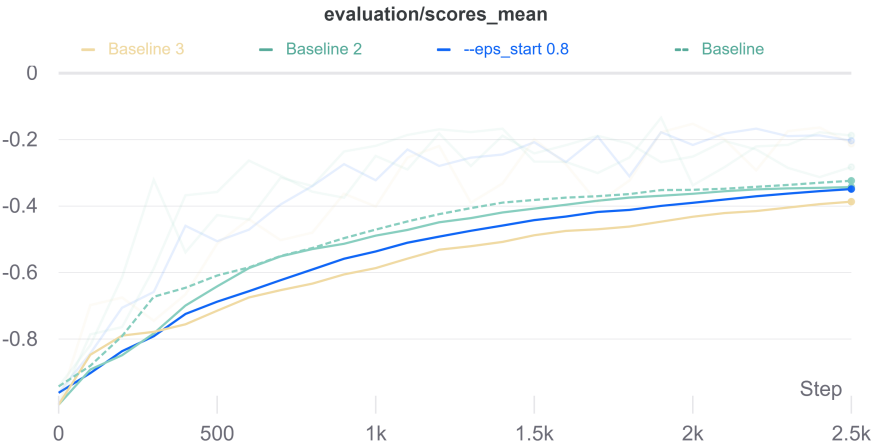

Figura 3.12: Score during evaluation episodes with epsilon= 0.8 at the start of the whole run

As shown in the graphic, there's no benefit on changing the value of epsilon at start.

epsilon_end = 0.1

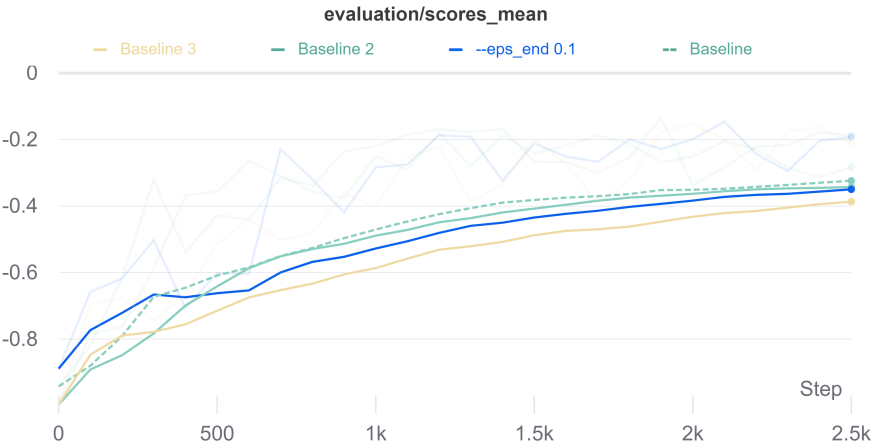

Figura 3.13: Score during evaluation episodes with 0.1 as minimum value for epsilon

As shown in the graphic, there's no benefit on changing the minimum value of epsilon.

epsilon_decay = 0.8/0.9

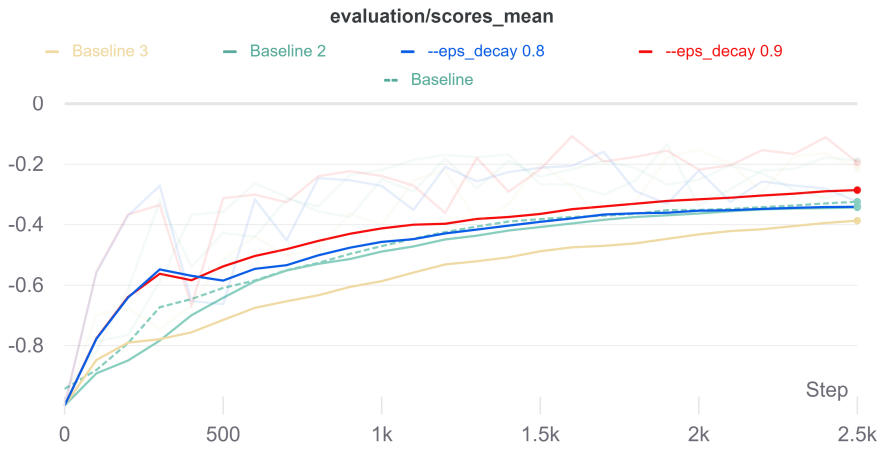

Figura 3.14: Score during evaluation episodes with 0.1 as minimum value for epsilon

It seems there is an improvement with a decay of 0.9 but the improvement is too little to say anything, as it is shown in the other graphics in the appendix.

Gamma Gamma = 0.85:

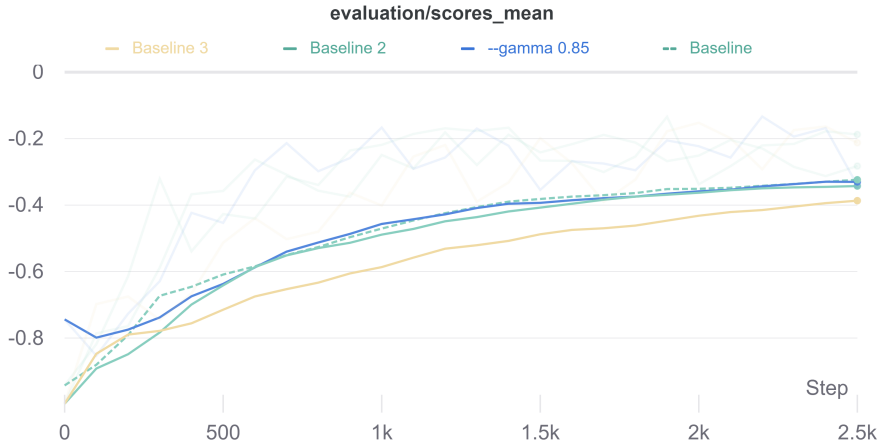

Figura 3.15: Score during evaluation episodes with gamma value of 0.85

As shown in the graphic, there is no benefit from reducing the value of gamma to 0.85.

Learning rate The test here are conducted with a learning rate two and ten times bigger than the default one (respectively 0.1e-3 and 0.5e-3). In the end, both performs the same as the baseline, but increasing the learning rate makes the model performs better with a lower number of episodes.

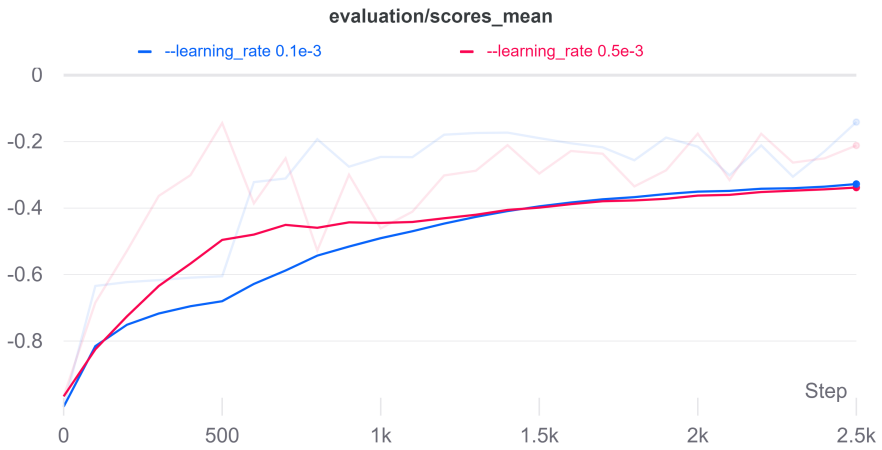

Figura 3.16: Score during evaluation episodes with learning rate of 0.1e-3 and 0.5e-3

But since we're interested on 2500 episodes per run, changing the learning rate doesn't effect the performance of the model.

Tau The test here are conducted with a tau value ten and twenty times bigger than the default one (respectively 1e-2 and 5e-1). In the end, both performs the same as the baseline, but increasing tau makes the model performs better with a lower number of episodes.

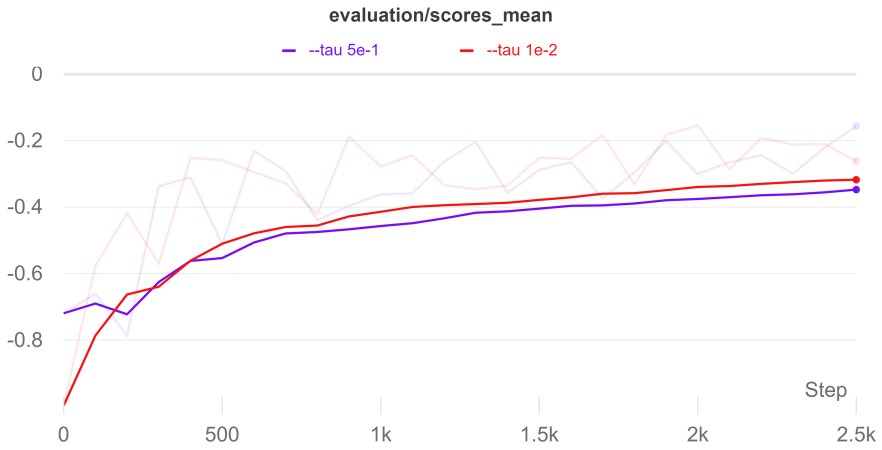

Figura 3.17: Score during evaluation episodes with a tau value of 1e-2 and 5e-2

But since we're interested on 2500 episodes per run, changing tau doesn't effect the performance of the model.

Conclusions Unfortunately, this phase of testing did not bring any particular result: so it is easier to keep the as default, to get the same results with a shorter query.

Capitolo 4

Reward

Flatland's benchmark already offers an implementation of rewards as described in chapter 2.3. According to these rewards, the agents learn that the more they move around, the worst is the reward (*step_penalty*), while if they reach the target they receive a positive reward (*global_reward*). This forces the agents to reach the targets and do it as fast as possible. Even if this simplistic reward system is enough for the agents to accomplish their tasks, something more could be done to fasten the learning process.

4.1 Approach discount

One idea is to penalize more the trains which goes completely random with respect to the ones that make some improvements, even if do not reach the target yet as suggested by [1]. The first step then, was to decrease the *step_penalty* in the case the agent is correctly moving towards its target. This was made by introducing an heuristic, the Manhattan distance, and, instead of having *step_penalty* as a constant, introducing a function that checks if the agent move is decreasing the Manhattan distance with the target. In case the distance is reducing, it means the train is going towards the target. In this case the penalty is reduced.

Limits Often it happens that the only track to the target does not necessary reduce the Manhattan distance at each step. In those "not optimal" steps, the penalty is full, but this would violate the principle of "do not penalize a train that is going towards the target". However, the intuition is still valid because in the default environment the penalty would have been greater.

For example:

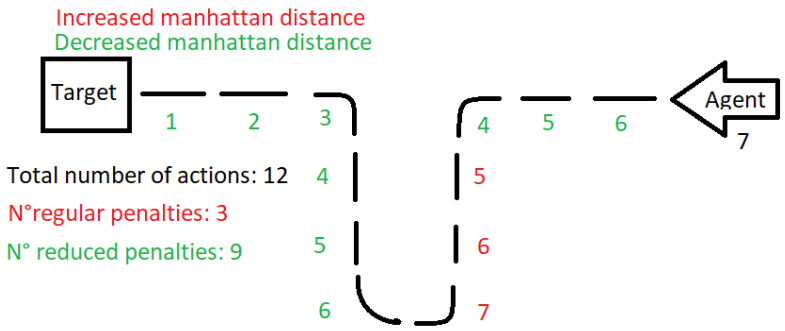

Figura 4.1: Example of the new reward system: even though there still are 3 regular penalties,it's a big improvement from the default reward system, that would have given 12 penalties

Here's the code developed:

```
def Manhattan_distance(self,pos1,pos2):
    return abs(pos1[0]-pos2[0])+ abs(pos1[1]-pos2[1])
```

The Manhattan distance is defined as the sum of the horizontal and vertical distance between two points: it can be thought as the space between two positions in a world where only horizontal or vertical steps are allowed, and that's the case for the Flatland environment.

```
def through_target(self, agent: EnvAgent):

    old_distance=0
    distance=0

    # if the agent has just been deployed
    if agent.position and not agent.old_position:

        return True

    # if the agent made at least one step
    elif agent.position and agent.old_position:

        old_distance = self.manhattan_distance(agent.old_position, agent.target)
        distance = self.manhattan_distance(agent.position, agent.target)


    if distance<old_distance:
        # if the agent is getting closer to the target
        return True


    else:
        # if the agent is getting farther from the target (or it hasn't been deployed)
```

```
            return False
```

The function *through_target* returns 'True' whenever the agent is getting closer, regarding its Manhattan distance, to the target. This value is then used to discriminate the reward to give to the agent: when *through_target* outputs True, the agent will receive a reduced reward.

```
def step_penalty(self, agent: EnvAgent):
    if self.through_target(agent):
        discount= 0.1
        return -discount * self.alpha
    else:
        return -1 * self.alpha
```

4.2 Deadlock discount

When trains are in a deadlock state, they will not be able to reach their target and will block other trains from moving on the blocked rail. This is a situation that must be avoided as much as possible. The idea suggested by [1] is to increase the step penalty for trains that are in a deadlock state.

This is implemented by looking every possible transition of every train and recursively check if all trains that block those rails are also blocked by other train. This will at the end find a subset of the trains that are all blocking each other.

Capitolo 5

Flatland Observations

The building of observations offered by the benchmark is quite inefficient, especially when the tree dept increase. We changed the way the tree is built in order to make it easier to create new features and more efficient. Also, we changed the observation to improve the performance of the agents.

5.1 Graph

In order to optimize and make it easier to create features, every time a new 2d map of the world is generated, we create a graph containing all useful information of the map as suggested by the work made by Jonas Wälter [4]. Every switch is encoded as a maximum of four different nodes in the graph, one from each possible arrival direction. The possible connections with other switches can indeed change depending from the arrival direction, and so do the graph nodes. For example, in the figure 5.1, the circled switch can be reached by 3 possible orientation (north, south and east). This by arriving to the switch facing north the trains can only keep going north, while a train which is going south can choose if going south or west and so on. The rail cells are then encoded by the parameter "distance" from one graph node to the other. To help the feature development we also created a map to associate every tuple (rail cell, direction) to the tuple (node, distance from the node). This association should encode which switch will be reached from a given rail cell and direction and how far is it. This graph is then used to reproduce the graph observation features more efficiently and to create new features in a simplest way.

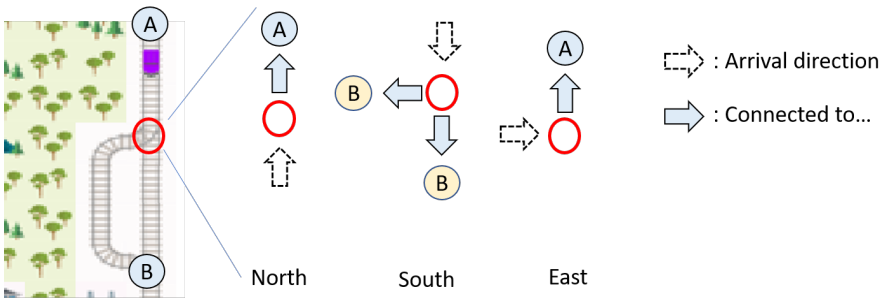

Figura 5.1: Switch decomposed into three nodes of the graph

5.2 Interpretation of tree node

In the benchmark implementation of the tree observation, a tree node is created every time there is a switch where the agent can take different path. This is probably meant to create a node every time an important decision could be taken from the train. This would exclude the *non usable switch* from being nodes of the tree observation (one example of *non usable switch* is the circled switch for the red train in figure 5.2). the information about *non usable switches* is encoded in a feature that gives for every node the distance from the first *non usable switch*. Anyway this could be too much a simplistic model and could exclude some information. One case where the lack of information could result critical can be seen in figures 5.2 and 5.3. The tho cases could not be distinguished even if the first one could be solved if trains takes the right paths while the second will cause a deadlock for sure.
Another possible definition of the tree node could then be to create a tree node every time there is a switch, regardless if the train can take only one or more paths. The idea is that even if the train cannot take any decision except from going forward and stop, other trains could. Also, stopping before a switch, can avoid to block other train which are going on the opposite direction, so it could be really important to get full information even for every *on usable switch*.

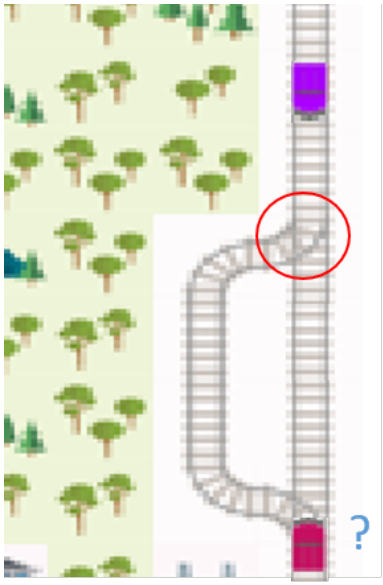

Figura 5.2: Example of a switch without choice (for the red train's point of view)

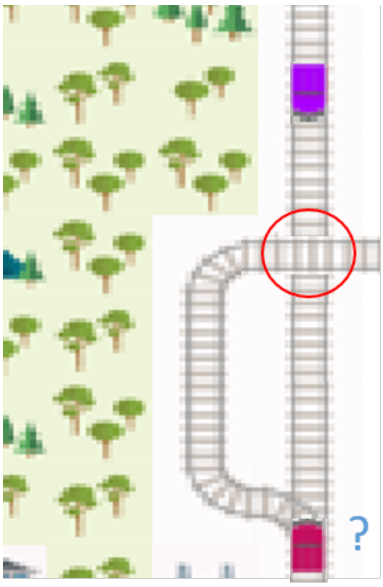

Figura 5.3: Example of a crossing switch

5.3 Deadlock

A negative situation can happen when the trains take a path which irreversibly brings to a deadlock. When two or more trains are in a deadlock state, they will not be able anymore to reach their target and can block a rail forever, making it more difficult for other trains to reach their destination. For those reasons deadlock should be avoided

as a first priority.

Given the agent's observation, the trains should be able to learn how to avoid deadlocks. Anyway, predicting deadlocks before they happen is not easy and not always possible using only the considered observation features. One case where is not possible for the train to predict the deadlock is the case showed in figure 5.2. In this case the red train has no way of knowing if the purple train will block it on the circled switch or not, as the exploration tree does not tell whether the purple train can only go south or not. For this reason the red train has no way of knowing if going left there will be a third train blocking the purple train.

The intuition is then to add a feature to the observation tree which tries to predict if the train will cause a deadlock by going there. The assumption is that other trains won't move before the current train will reach the considered node. In practice, by the time the considered train will reach the node, the situation could be changed. Anyway, it is still a measure of a risk of deadlock and suggests which path should be avoided. It is then a responsibility of the reinforcement learning algorithm to learn that if the node is far from the agent, then the situation could change.

Implementation The implemented algorithm detects deadlocks only considering the node and its neighbors so while it guarantees that if it detects deadlock there must be a deadlock, it will not guarantee that if no deadlock is detected then there cannot be any deadlock. To find out if there is a deadlock on a node given the input rail (where the considered train is coming from), first all deadlock inherent nodes are detected. Those nodes are labeled as inherent nodes if are directly connected to the input rail or if are directly connected to another inherent rail. Considering the previous example, the pink train will detect the left rail as deadlock inherent rail and explore it to determine if also the red train is blocked or not (figure 5.4 shows how are detected the inherent rails).

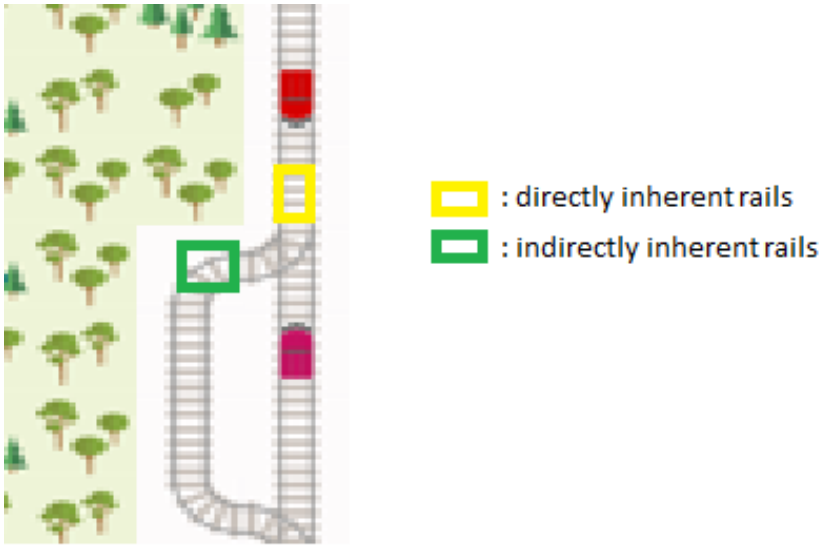

Figura 5.4: Deadlock inherent direction labeled according to the point of view of the pink train

Then, there a deadlock is detected on the node if one of the two cases are true:

* Another train is already inside the considered node, coming from the opposite direction.
* There is a train coming to the node from every deadlock inherent node

Capitolo 6

Neural Network

In this chapter it is shown a possible modification to the neural network and its application in the reinforcement learning process

6.1 Convolution neural network

Convolution neural networks are really powerful techniques which permit a better and faster training compared to classical fully connected neural layers. Anyway, convolution neural networks can only be used under strict conditions. In this problem, the input data used by the Dueling Double DQN is structured and hierarchical, so instead of using simple fully connected neural layers, something different could be done.

Convolution as data reduction The data is organized by a series of nodes where for each node some features are available. We exploited this structure by applying a convolution 1D with both kernel and stride equal to the number of features. The idea is that in this convolution the network will first analyze the node features node by node and summarize them into a more elaborated ones. Then those new features can be compared using the next layers of the network. This can be seen as a dimension reduction problem where the dimension of the input is reduced and only important information for the network are kept. Figure 6.1 shows a simple case of convolution used to reduce the feature number for each node of the tree. The tree is first transformed into a string of data, then the convolution is applied as described before.

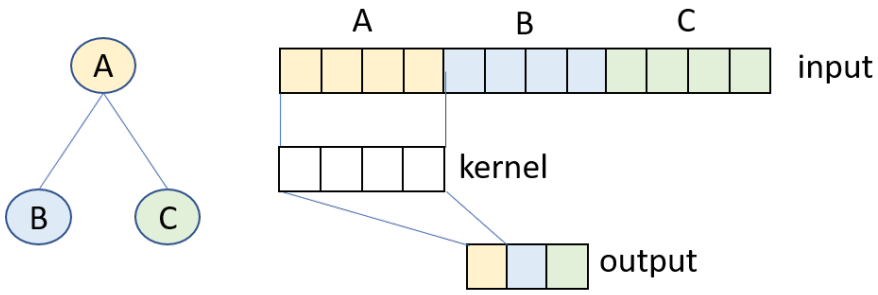

Figura 6.1: Convolution used to reduce the number of feature per node

This convolutionary layer could either be followed by the two linear layers used by the benchmark solution or it could substitute the first linear layer and have only one linear layer after it. The first case can be seen in the schema 6.2. We have chosen to try both alternatives and compare the results.

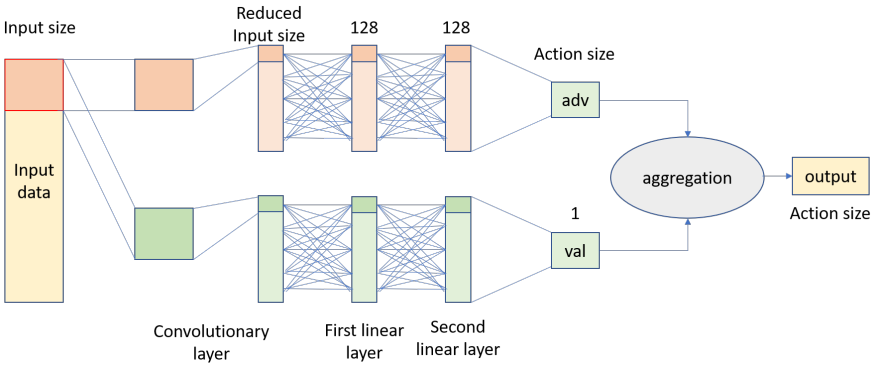

Figura 6.2: Convolution added to DDDQN schema

6.2 Legal actions

The neural network allows to select any action in any situation, even those that does not make any sense. For example, an agent cannot perform the Move Right actions at all if there is no track connection to the right. Those actions are referred to as invalid actions and in order to optimize learning, an agent is prevented from selecting an action that is invalid in the current situation.

6.3 Action required

An analysis of the rail network has shown that there are basically only two situations in which a decision should be taken: before a fork or before a join. The first situation means that the agent can end in two different cells from his current position and orientation (figure 6.3, on the left). While the second one, means there is a path that join the current track of the agent (figure 6.3, on the right). In both situation the agent can take more than one action so the neural network should be used to define which action the agent should take.

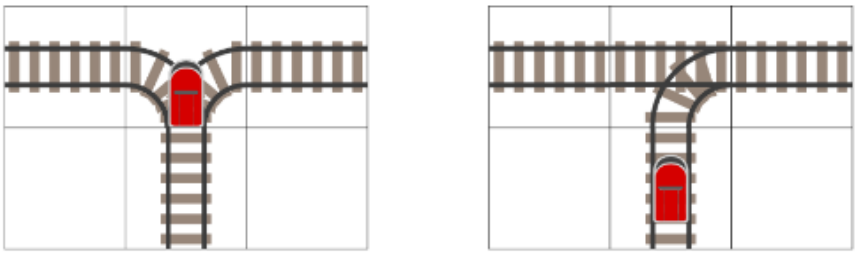

Figura 6.3: Situation in which the neural network should be used to take a decision

In any other situation there is only one possible action. Thus, in those situations (as the one shown in figure 6.4), it does not really make sense to use the neural network to decide which action to choose and it should be chosen the action which lead to the next cell. In this way we can reduce the number of application of the neural network.

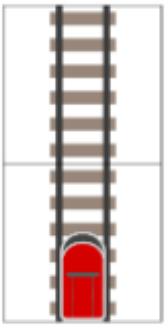

Figura 6.4: Situation in which there is only one possible action for the agent and thus it should be performed

6.4 Output size

The output of the neural network is needed to train the neural network itself. Indeed, it is reasonable to think that the more the possible actions are, the more the network has to learn. For this reason we tried to reduce the size of the neural network, without reducing the actual number of actions that the agent can make.

Do nothing In order to reduce the output size we have noticed that the action *Do Nothing* is not needed. Contrary from the other actions, it does not give any direct movement instruction but it is used to repeat the previous action. Therefore, the action is redundant and can be removed to decrease the complexity of the output.

Action mapping So far, the output of the neural network corresponds to the action size of an agent. By looking at the environment, an agent can always decide between at most two cells in addition to stop on the current cell. We can therefore further reduce the output size by removing one possible output. In this way, the output no longer corresponds directly to the actions and it should be mapped as shown in figure 6.5.

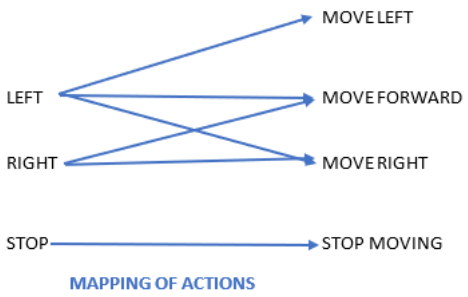

Figura 6.5: neural network output mapped to the action

Capitolo 7

Results

In this chapter the results of all the modification are shown. In order to keep tests coherent and comparable we used the same hyper-parameters in particular we used the default ones for the reasons described in chapter 3.

7.1 Rewards

We tested two alternatives: the change of regular penalty in case the trains get closer to their target and the penalty increased in case of deadlock.

Regular penalty By lowering the penalty in case the agent is running towards the target the performances increase. Here the discount variable has been tested with different values (0.1, 0.2 and 0.55) on the environment 0 (default environment) and on the environment 1 which are described in chapter 3.2. As can be see in figure 7.1 and 7.2, the difference between discount variable of 0.1, 0.2 or 0.55 is not significant. On the other hand, all three cases preform better that the baseline both with environment 0 and environment 1.

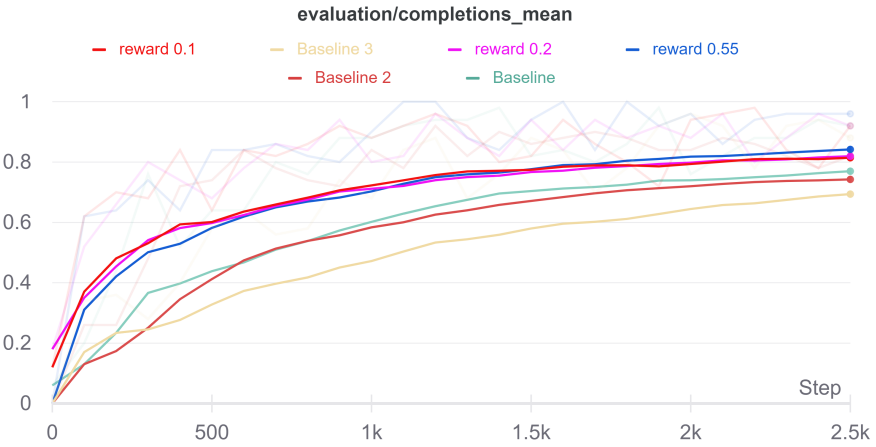

Figura 7.1: Discount variable of 0.1, 0.2 and 0.55 compared with baseline on environment 0 (default)

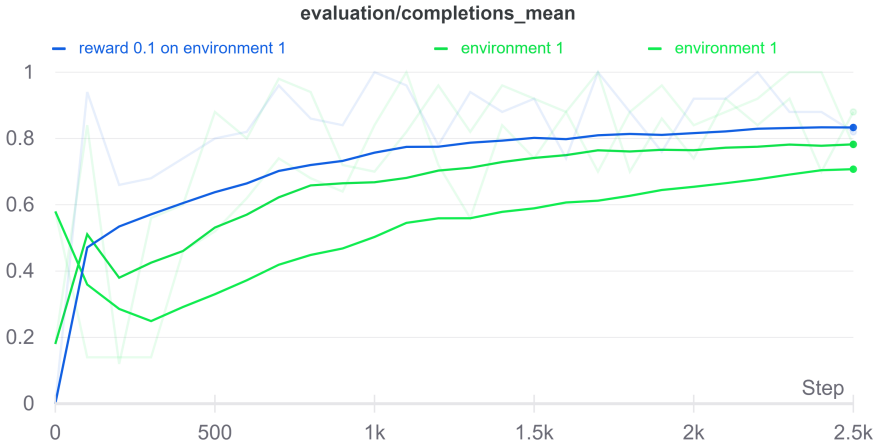

Figura 7.2: Discount variable of 0.1 compared with baseline on environment 1

Deadlock penalty We then tested the increased penalty in case of deadlock. For the test we choose to give a penalty of -2 (instead of -1). Anyway the results did not change with the respect to the baseline.

7.2 Graph observation

First, we tested the agents with both the benchmark implementation and our implementation using graphs created by us, but still without adding the feature for deadlocks. The result of the two tests does not have any evident difference. The observation tree should e about the same in both cases so this result is quite expected. It shows anyway that there are no big mistakes in our implementation, otherwise there would have been a worsen of performance. One important aspect anyway is the time needed to create the observation for each agent, for each step. With an observation tree with max depth of 2, we could measure that with their implementation of the tree observation it takes on average 0.29 seconds, while in our implementation 0.31. This does not seem a big change, anyway by increasing the depth of the tree to 3 their implementation takes 0.55 seconds on average, while ours only 0.37. This can suggest that some operation that we make before exploring the possible paths could take lots of time, so the improvements can be only seen with an increasing tree depth.

7.2.1 Deadlock

We compared the benchmark results with the ones obtained by adding to the tree observation the feature deadlock which tells if there will be a deadlock in a node as explained in chapter 5.3. As shown in the graph 7.3, the number of completion does not increase but not decrease either. On the other hand, the graph in figure 7.4 shows, accordingly with the previous paragraph (section 7.2), that the time needed using graph observations with the addition of the deadlock feature considerably decreased, even with a tree depth of two.

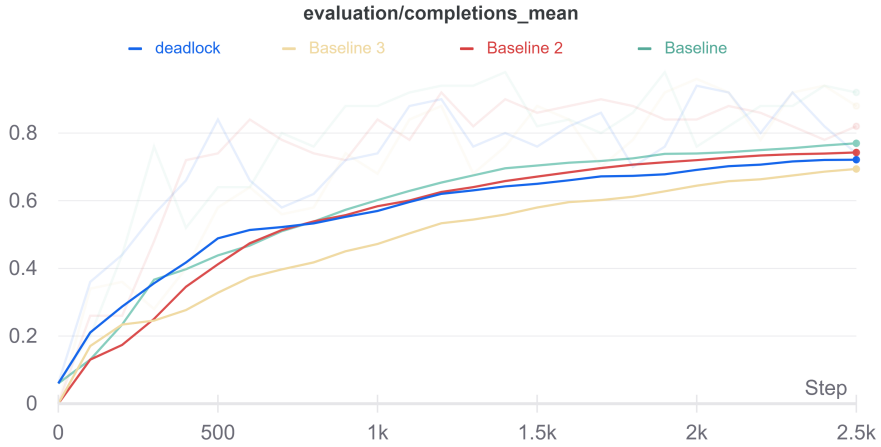

Figura 7.3: Completion mean using graph observations and deadlock feature

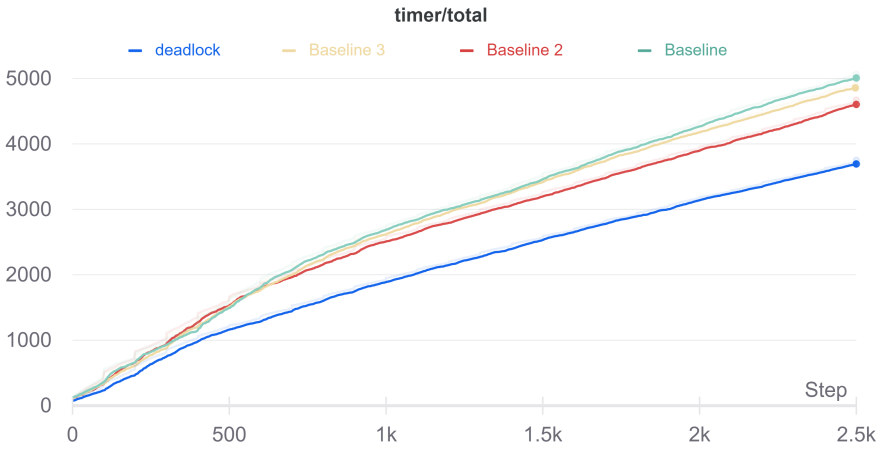

Figura 7.4: Time needed using graph observations and deadlock feature

7.3 Neural network improvements

At this point, we considered our observation as the baseline for the new comparisons. We compared our baseline with some improvement we have made to the neural network. In particular we add the improvement proposed in the chapter 6.3 and 6.4. First of all, used the neural network only when there is a real decision and the only possible action in any other case. Then we have prevented the agent from selecting an invalid action in the current situation. Finally, we have reduced the action by removing *Do Nothing* first, and then mapping the output of the neural network to the one of the environment. We compared every attempt with the baseline and the results are quite disappointing. Indeed, the average number of train who reach their target is 30% less than our baseline.

7.3.1 Convolution

Once again, we compared the results with or without using convolution but before doing so, we first found the best version of convolution network. To do so we first compared convolution followed by one or two other linear layers as explained in chapter 6.1. There was no visible change between the two models so we kept the convolution model followed by two linear layers. Then we tuned the number of channels created by the convolution neural network. The result is similar except when we used one channel of output. As can be seen in figure 7.5, the completion mean grows a lot faster in the first 500 episodes. Also the score, as shown in figure 7.6, behave in the same way. This suggests that maybe, as with only one channels the networks becomes less complicated, the training process fasten and the algorithm achieve the results faster. Anyway after a complete training (2500 episode) the result does not differ from other cases.

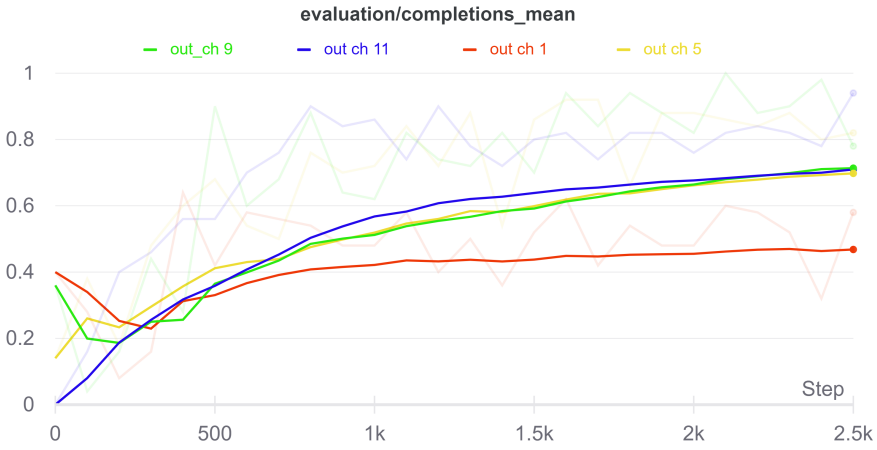

Figura 7.5: Completion mean using convolution model with different output channels

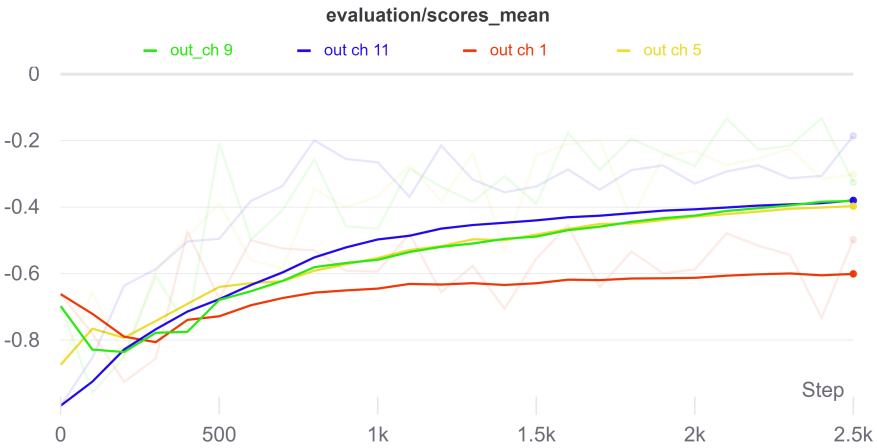

Figura 7.6: Scores mean using convolution model with different output channels

Finally, we compared the results of convolution model with the baseline. Unfortunately there is no difference in the results as shown in figure 7.7.

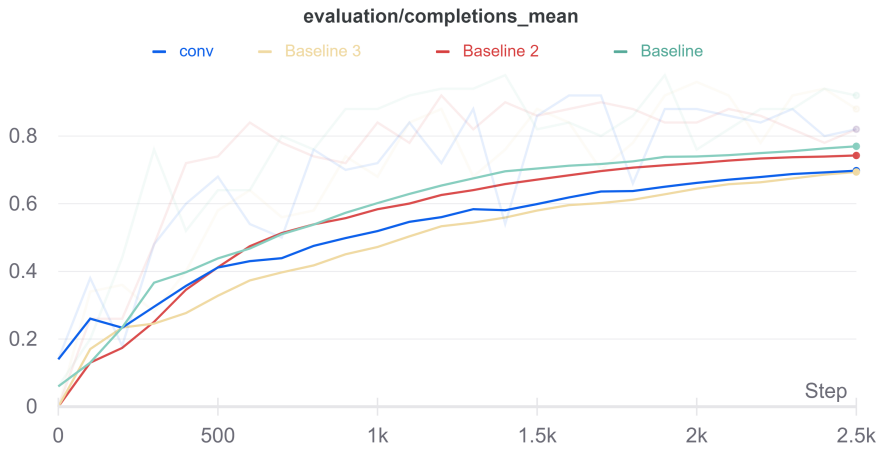

Figura 7.7: Completion mean compared between baseline and convolution model

Capitolo 8

Conclusion

In this work different improvements and solutions are investigated. As result, only few of them have been proven to be successful. We confirmed that the agents learn better when the reward function prize more the agents that get closer to their target. Also, even if it did not increase the percentage of completion for the agents, by changing the implementation of the observation we managed to decrease the time used by the whole algorithm. Finally, other modifications have been tried but brought none or bad results which is still helpful to exclude unproductive developments for future works. In particular, adding the deadlock feature to the observation tree and using a convolutionary layer does not sensibly change the results while other neural network's improvements brought negative results.

Capitolo 9

Future works

More improvements and variations are still left to be tried in future works. We propose some ideas that could be developed and tested in future.

* **Convolution neural network:** in this work was only tested a first layer of convolution neural network. Even if it did not bring the expected results, this path could be further explored.

* Rail cells: cells that permit the trains to pass over them.

Convolution In order to further develop convolution, convolution layers could be added. In this case the concept of neighborhood of 2d pixels are transformed into neighborhood in a tree. A convolutionary level can be applied with a kernel which include every node with its child's nodes. More convolutionary layers can be created as such, followed by a linear fully connected layer.
One possible implementation can be done by creating redundant data and placing each node close to its children, then a 1d convolution can be applied with Kernel size and stride equal to size(node) * (1 + children number). The output of this convolution is then a sub-tree, so another convolution equals to this one can be then applied recursively by duplicating the output of this layer in the same way as it is explained above. The redundant data could be misinterpreted as an increasing of parameters for the neural network and so an increase of time for learning. Instead, in convolution neural networks the number of parameters depends on the size of the kernel and not on the input size. Compared with a fully connected layer is a considerably small number. Picture 9.1 show a simple case of convolution applied to each sub-tree. The tree is first transformed into a string of data and some data is duplicated to keep convolution more simple (in this case node B and C are duplicated), then the convolution is applied for every sub-tree: (A, B, C), (B, D, E) and (C, F, G).

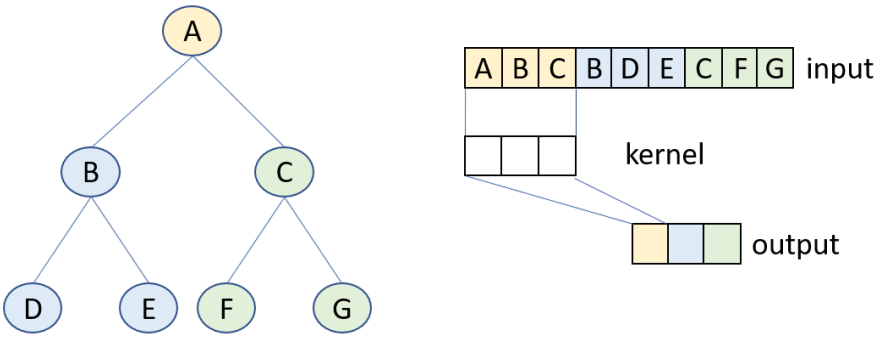

Figura 9.1: Convolution applied to each sub-tree

Appendice A

Appendix

Here all the graphics produced during the project are exposed.

A.1 Baseline

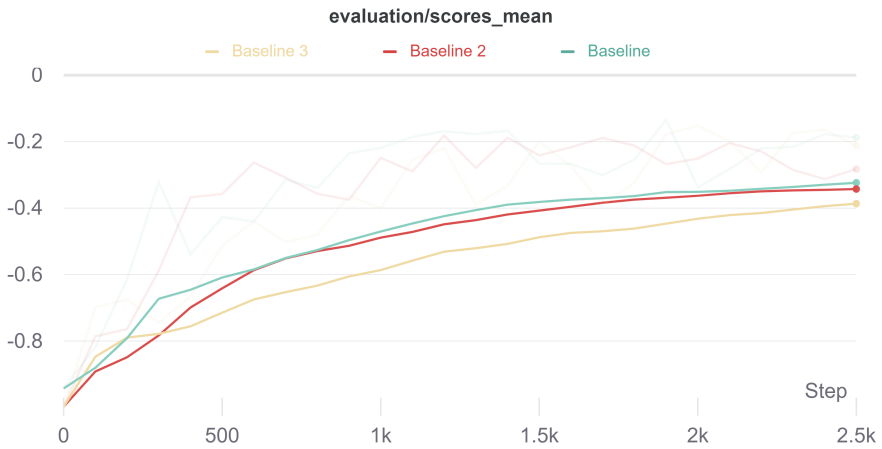

Figura A.1: Baseline: score during evaluation episodes

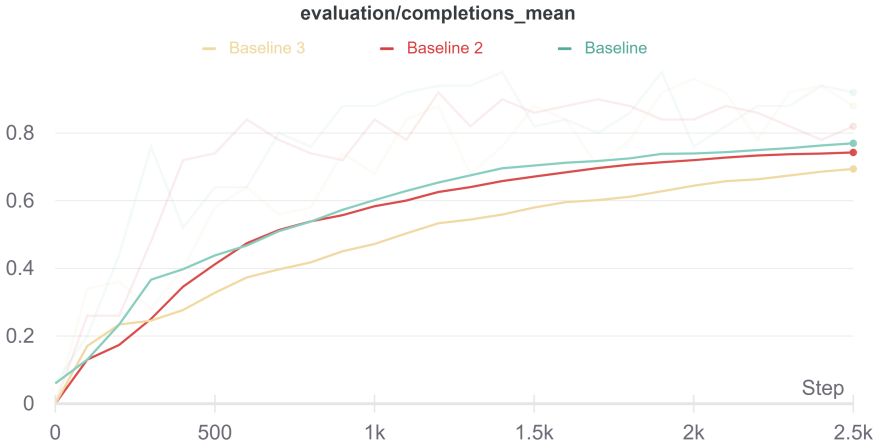

Figura A.2: Baseline: completion during evaluation episodes

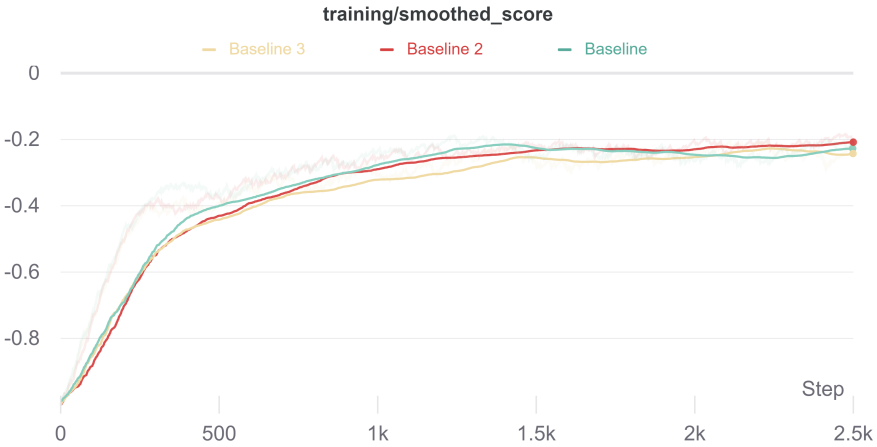

Figura A.3: Baseline: score during training episodes

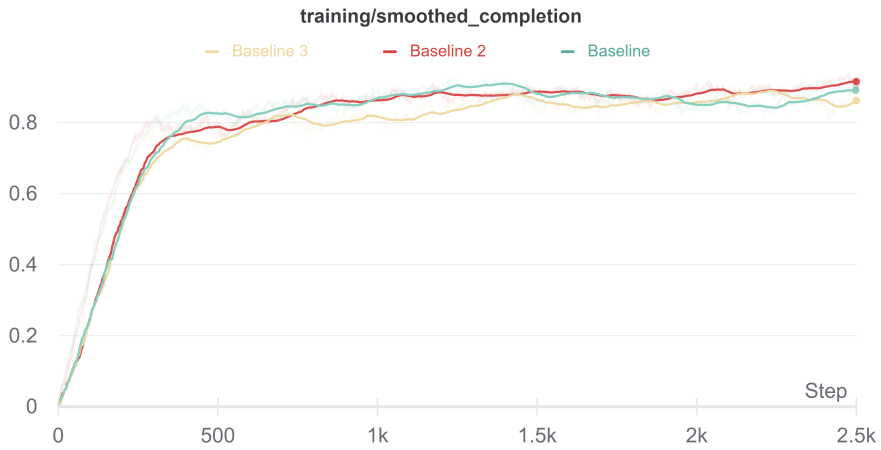

Figura A.4: Baseline: completion during training episodes

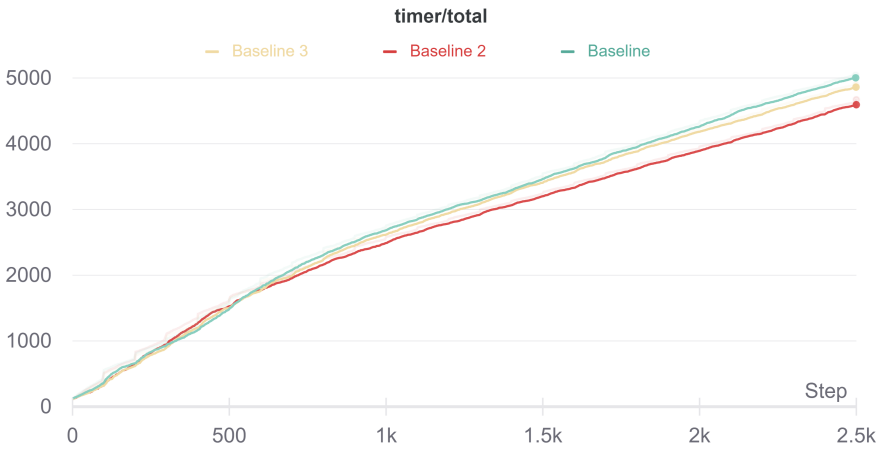

Figura A.5: Baseline: Timer of the execution of the run

A.2 Hyperprameter tuning: Training environments

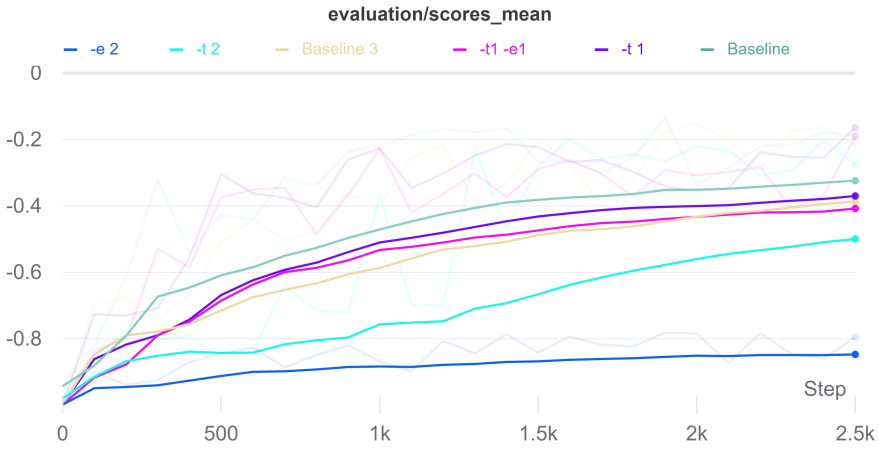

Figura A.6: Training environments : score during evaluation episodes

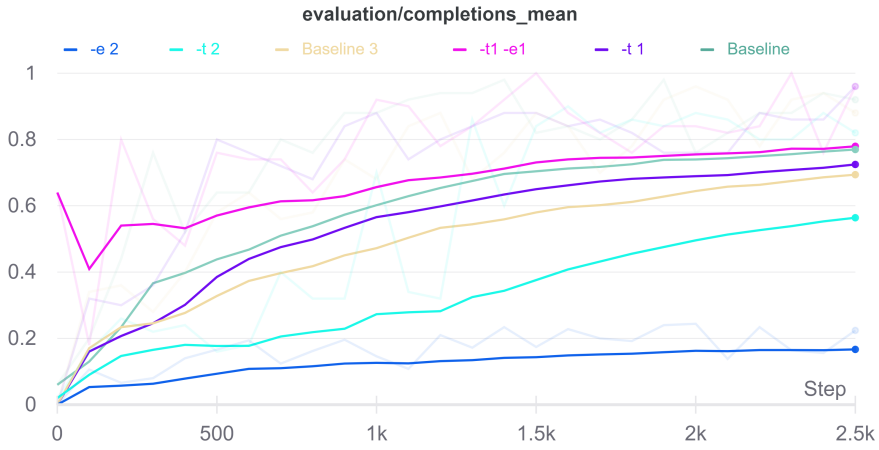

Figura A.7: Training environments: completion during evaluation episodes

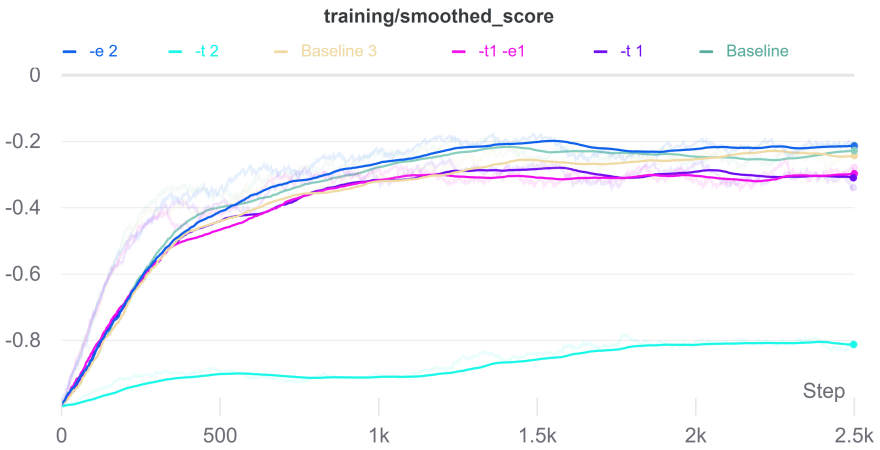

Figura A.8: Training environments: score during training episodes

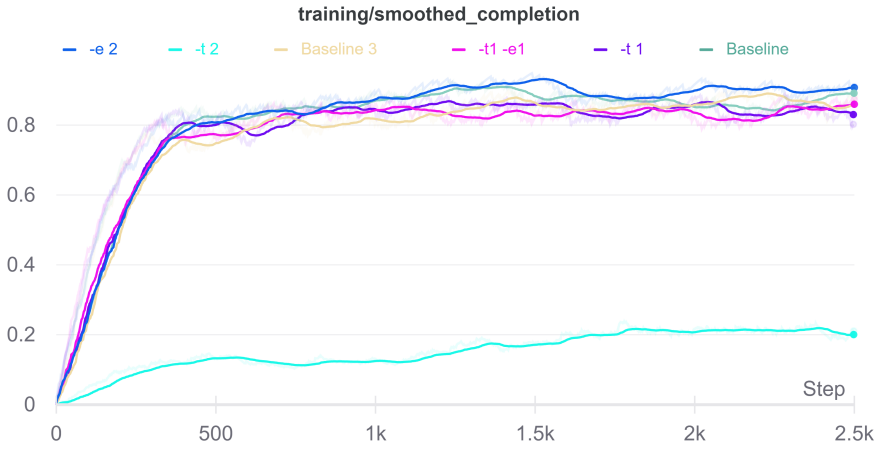

Figura A.9: Training environments: completion during training episodes

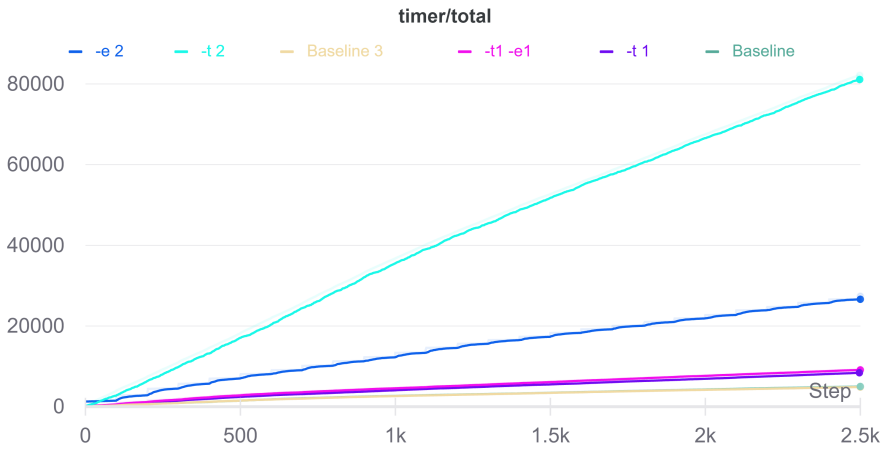

Figura A.10: Training environments: Timer of the execution of the run

A.3 Hyperprameter tuning: Epsilon start 0.8, epsilon end 0.1

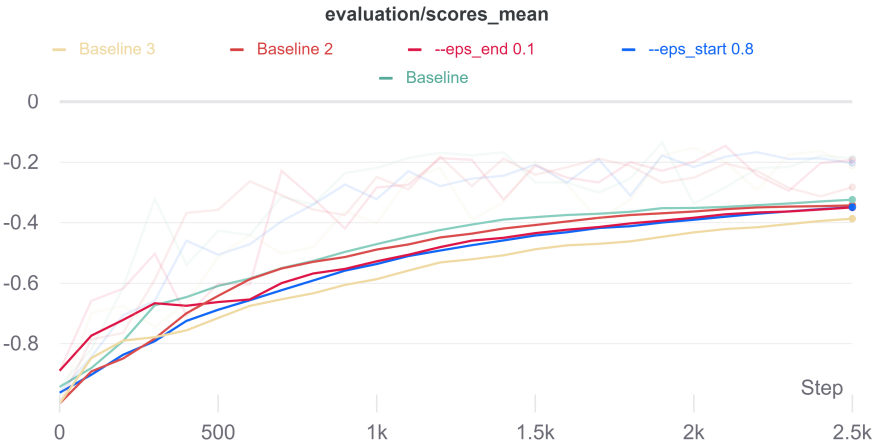

Figura A.11: Epsilon start 0.8 vs end 0.1: score during evaluation episodes

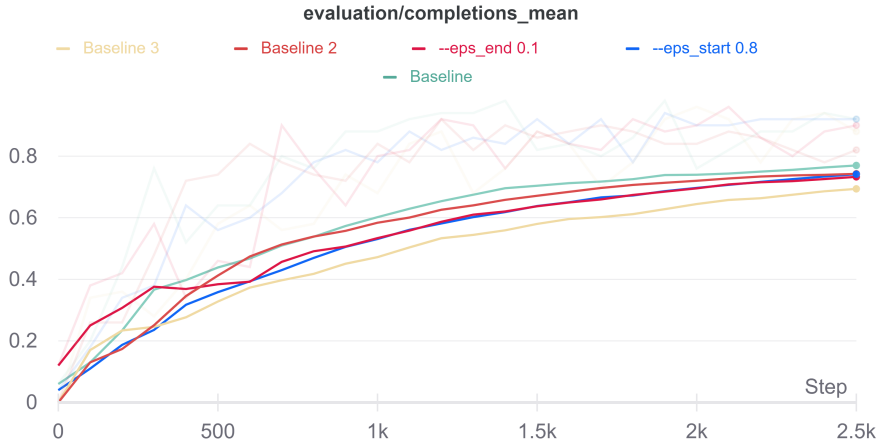

Figura A.12: Epsilon start 0.8 vs end 0.1: completion during evaluation episodes

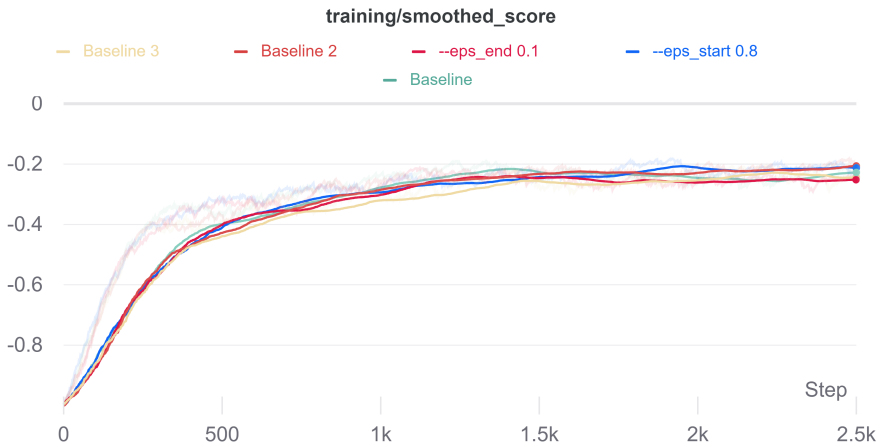

Figura A.13: Epsilon start 0.8 vs end 0.1: score during training episodes

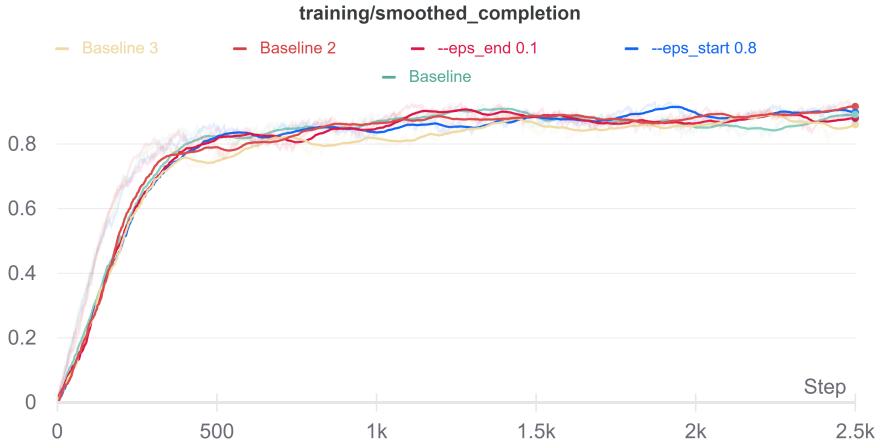

Figura A.14: Epsilon start 0.8 vs end 0.1: completion during training episodes

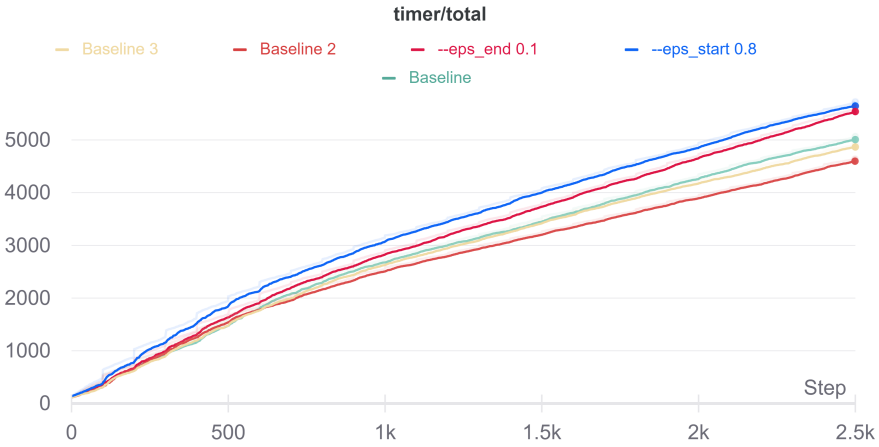

Figura A.15: Epsilon start 0.8 vs end 0.1: Timer of the execution of the run

Figura A.16: Epsilon start 0.8 vs end 0.1: value of epsilon over the episodes

A.4 Hyperprameter tuning: Epsilon decay 0.9 and 0.8

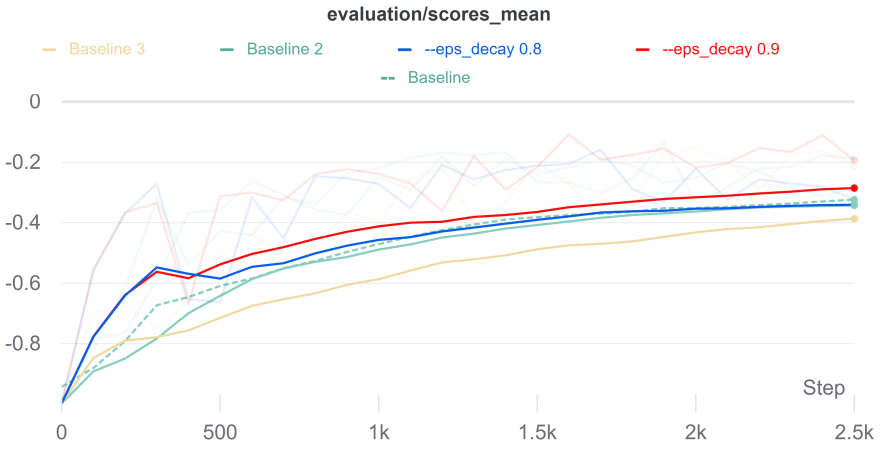

Figura A.17: Epsilon decay 0.9 vs 0.8: score during evaluation episodes

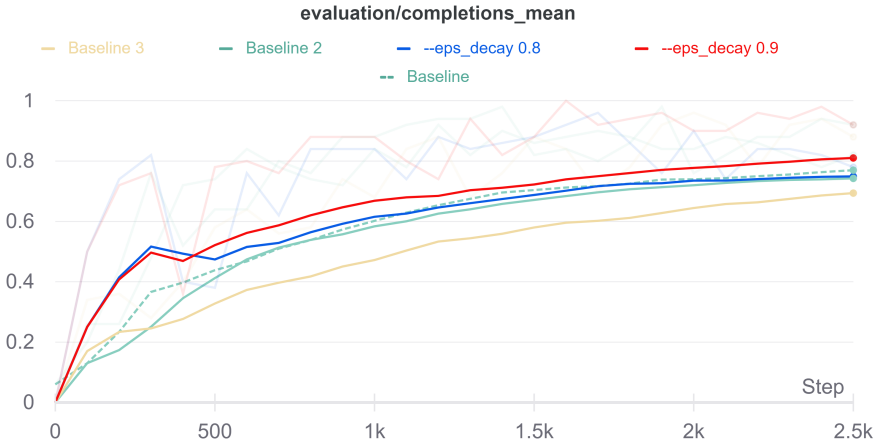

Figura A.18: Epsilon decay 0.9 vs 0.8: completion during evaluation episodes

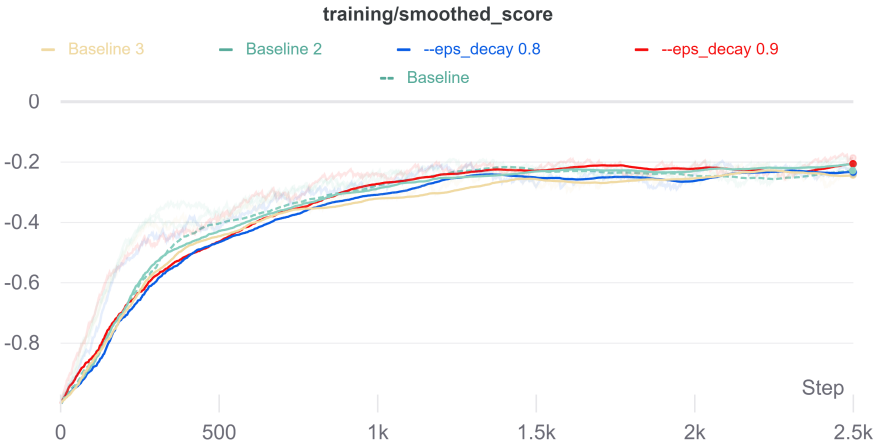

Figura A.19: Epsilon decay 0.9 vs 0.8: score during training episodes

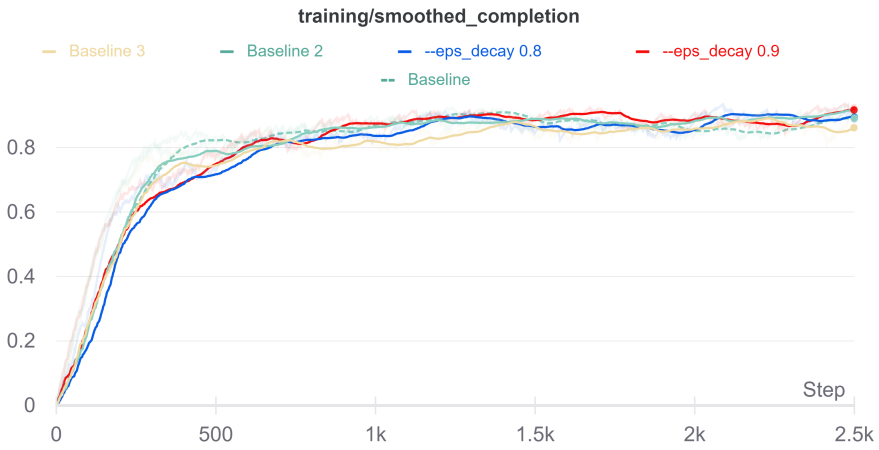

Figura A.20: Epsilon decay 0.9 vs 0.8: completion during training episodes

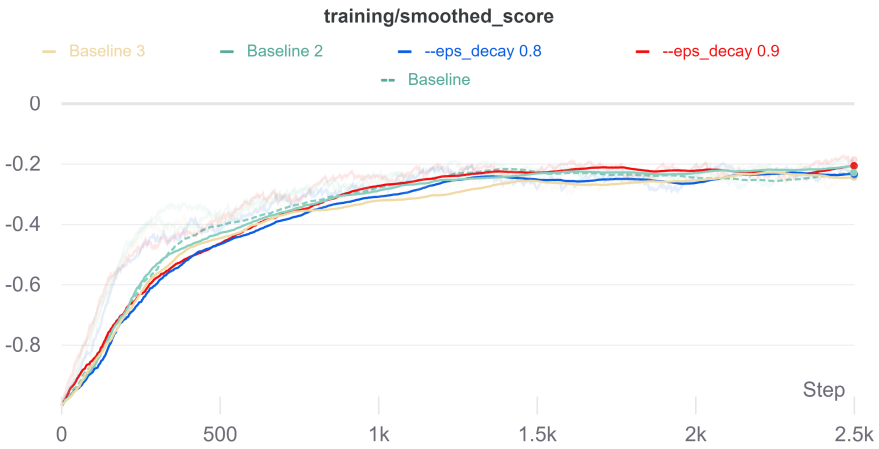

Figura A.21: Epsilon decay 0.9 vs 0.8: Timer of the execution of the run

A.5 Hyperprameter tuning: Gamma 0.85, Tau 1e-2 and 5e-1

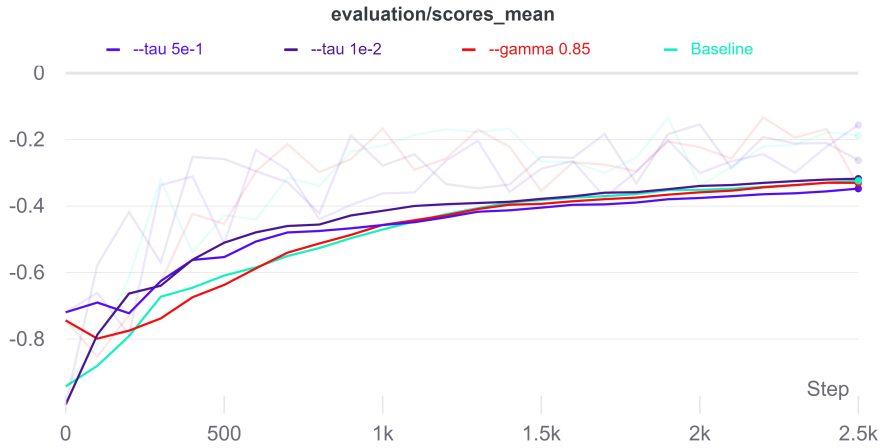

Figura A.22: Gamma 0.85, Tau 1e-2 and 5e-1: score during evaluation episodes

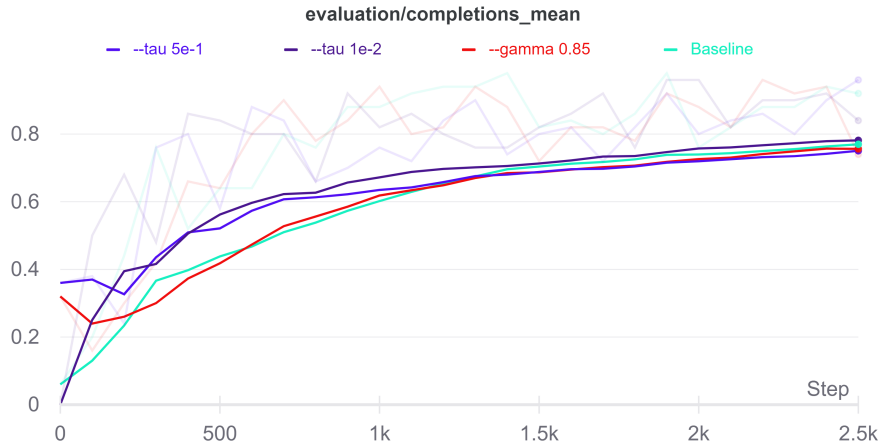

Figura A.23: Gamma 0.85, Tau 1e-2 and 5e-1: completion during evaluation episodes

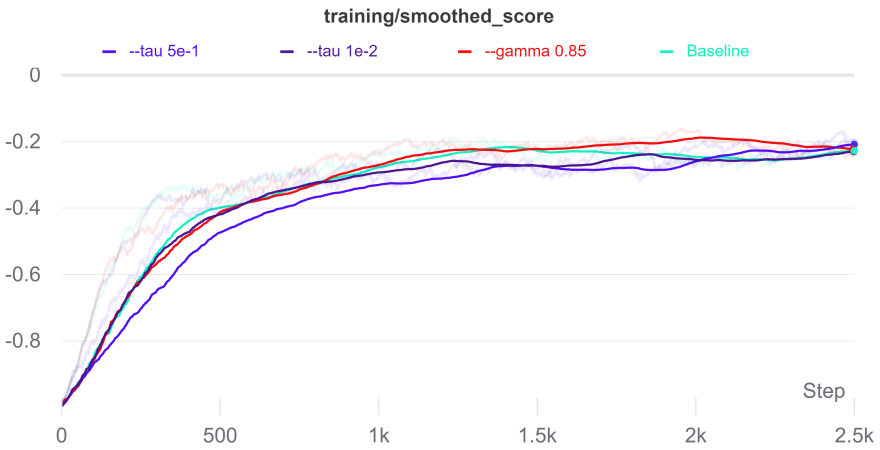

Figura A.24: Gamma 0.85, Tau 1e-2 and 5e-1: score during training episodes

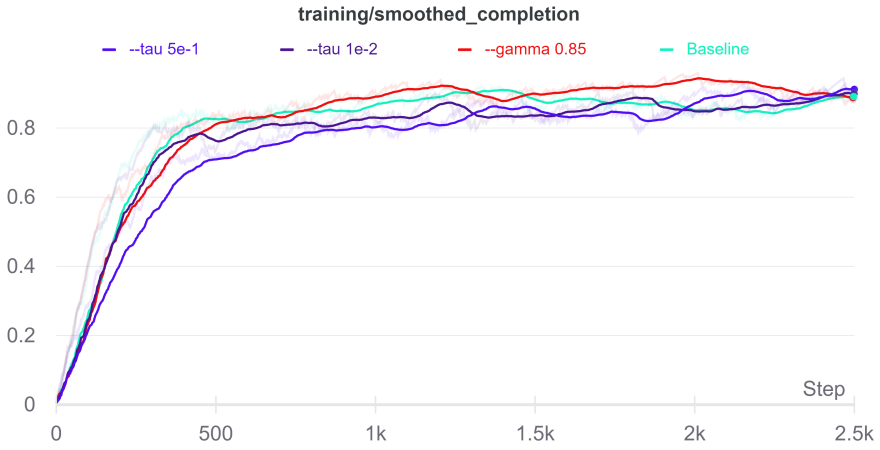

Figura A.25: Gamma 0.85, Tau 1e-2 and 5e-1: completion during training episodes

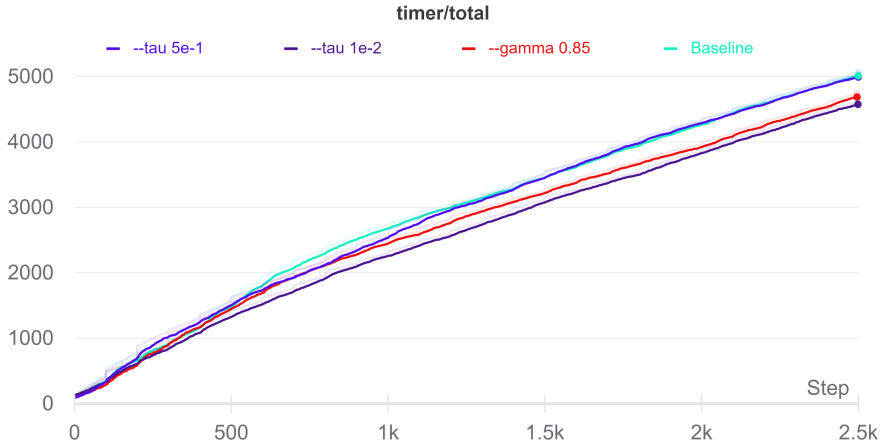

Figura A.26: Gamma 0.85, Tau 1e-2 and 5e-1: Timer of the execution of the run

A.6 Hyperprameter tuning: Learning rate 0.5e-3 and 0.1e-3

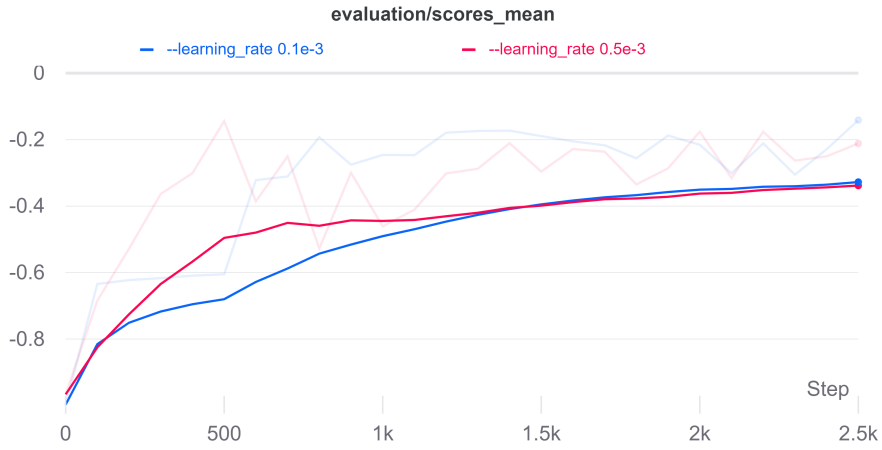

Figura A.27: Learning rate 0.5e-3 and 0.1e-3: score during evaluation episodes

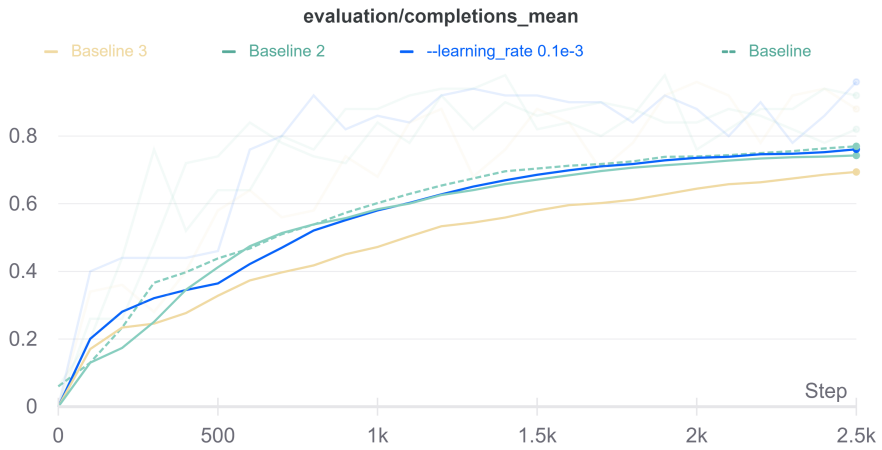

Figura A.28: Learning rate 0.5e-3 and 0.1e-3: completion during evaluation episodes

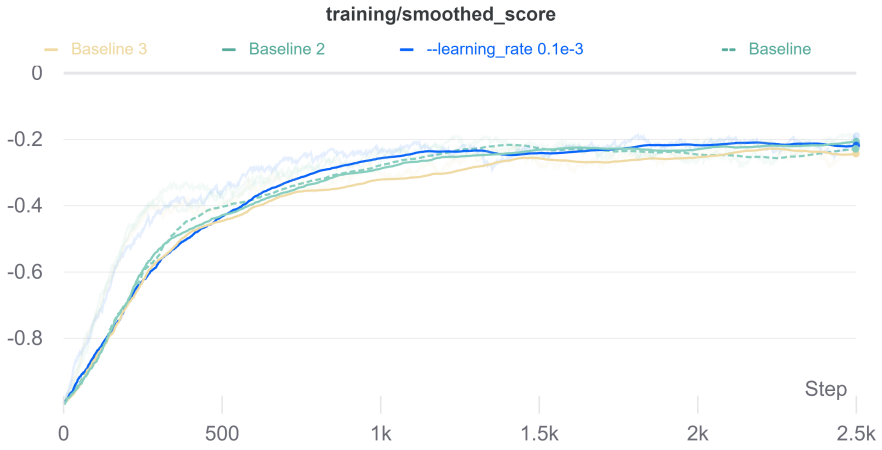

Figura A.29: Learning rate 0.5e-3 and 0.1e-3: score during training episodes

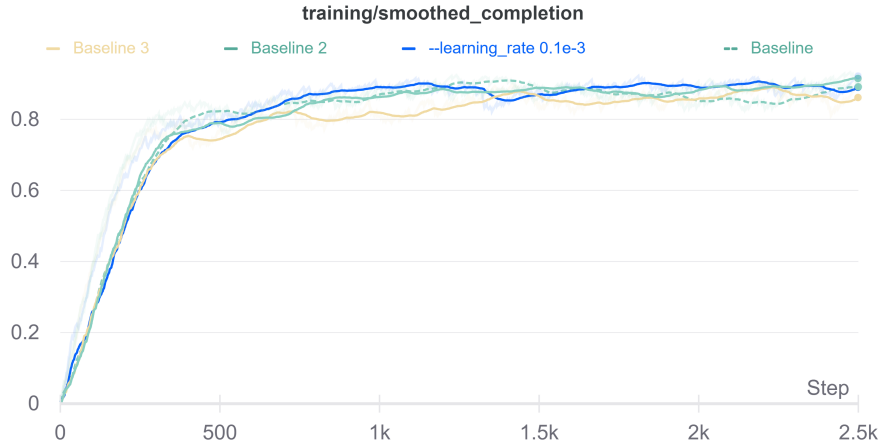

Figura A.30: Learning rate 0.5e-3 and 0.1e-3: completion during training episodes

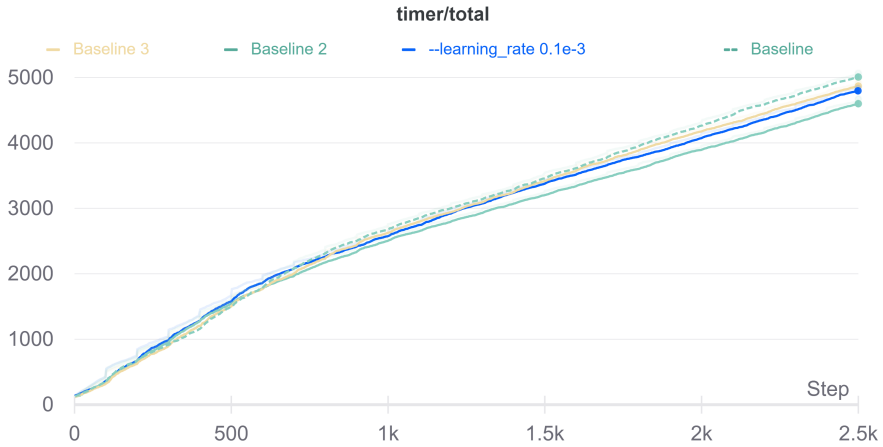

Figura A.31: Learning rate 0.5e-3 and 0.1e-3: Timer of the execution of the run

A.7 Out channels:Tuning

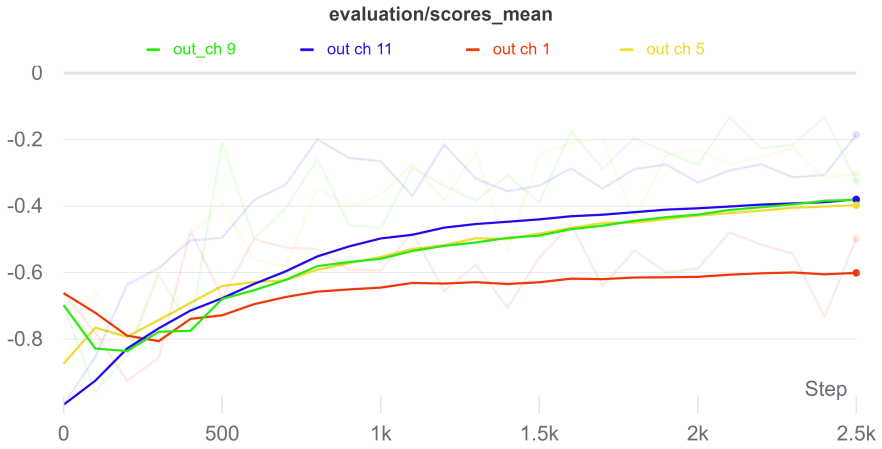

Figura A.32: Out channels tuning: score during evaluation episodes

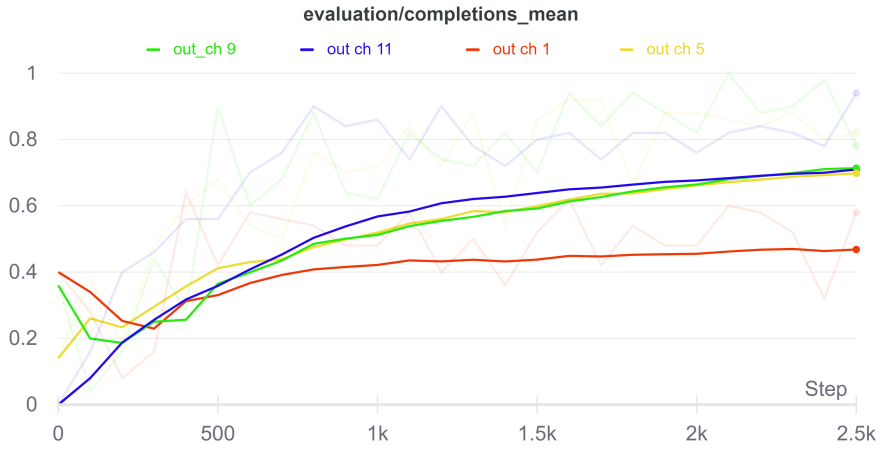

Figura A.33: Out channels tuning: completion during evaluation episodes

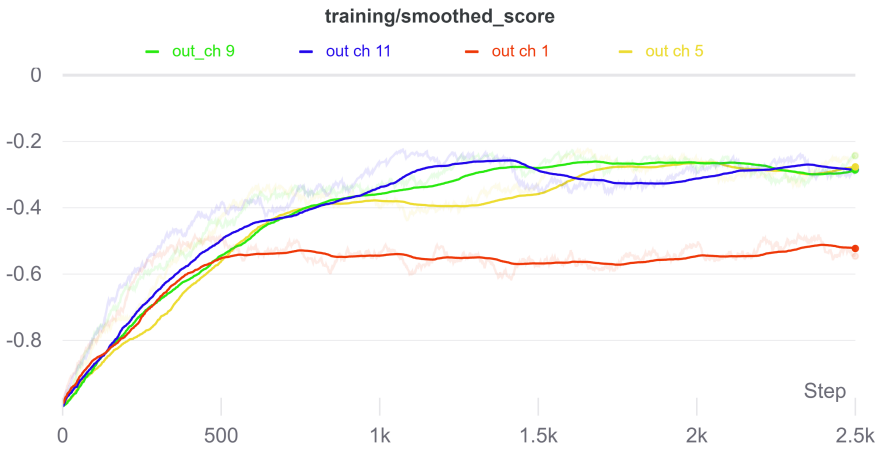

Figura A.34: Out channels tuning: score during training episodes

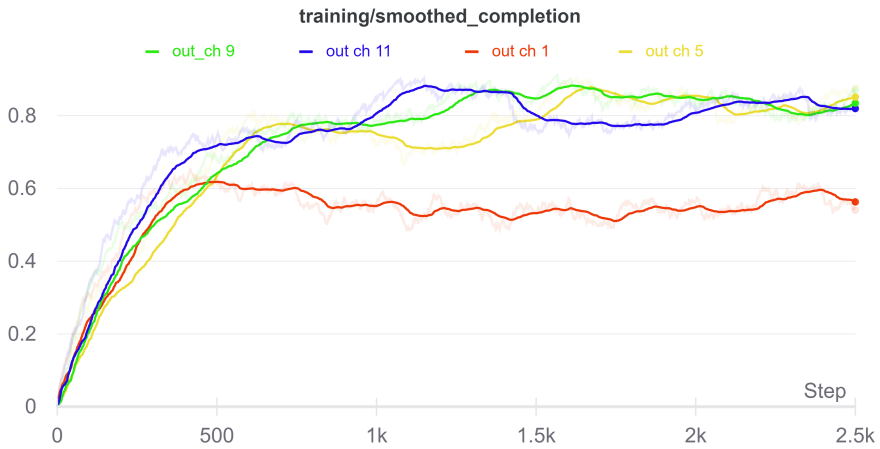

Figura A.35: Out channels tuning: completion during training episodes

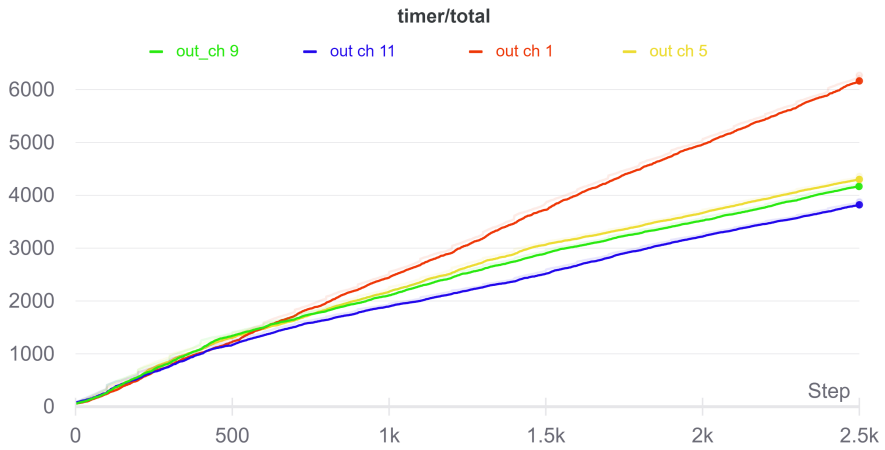

Figura A.36: Out channels tuning: Timer of the execution of the run

A.8 Convolutional network and simple convolutional network

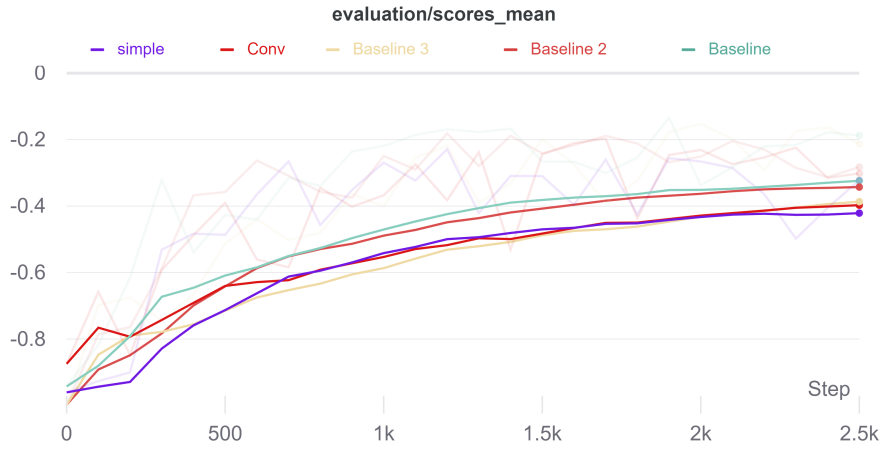

Figura A.37: Convolutional networks: score during evaluation episodes

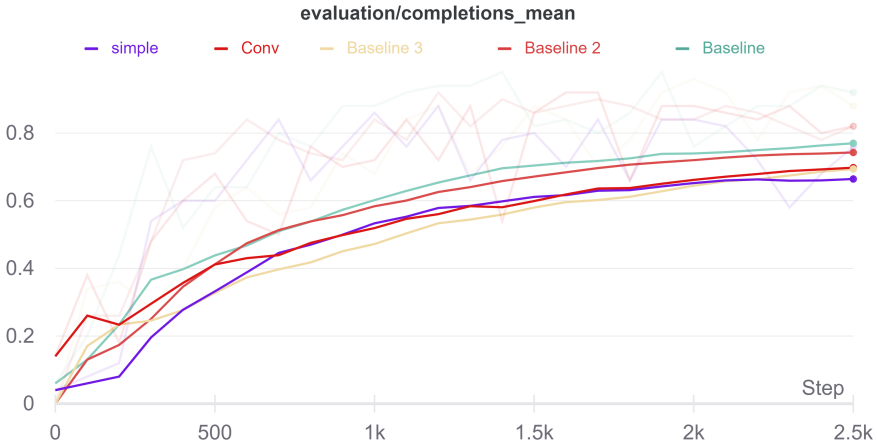

Figura A.38: Convolutional networks: completion during evaluation episodes

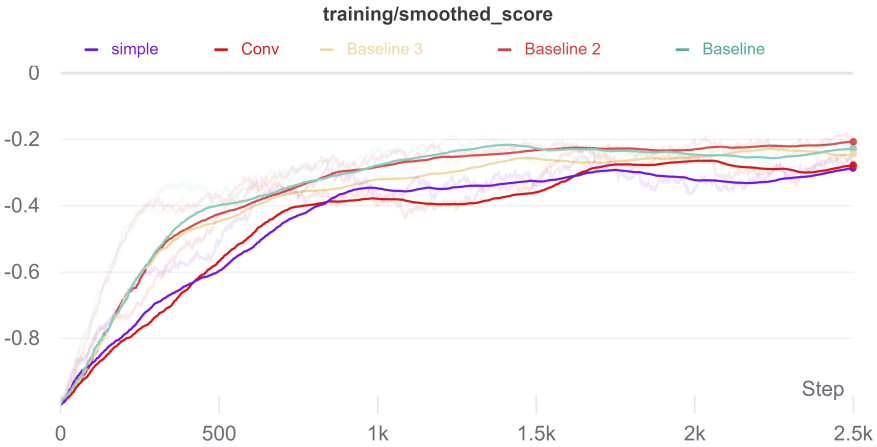

Figura A.39: Convolutional networks: score during training episodes

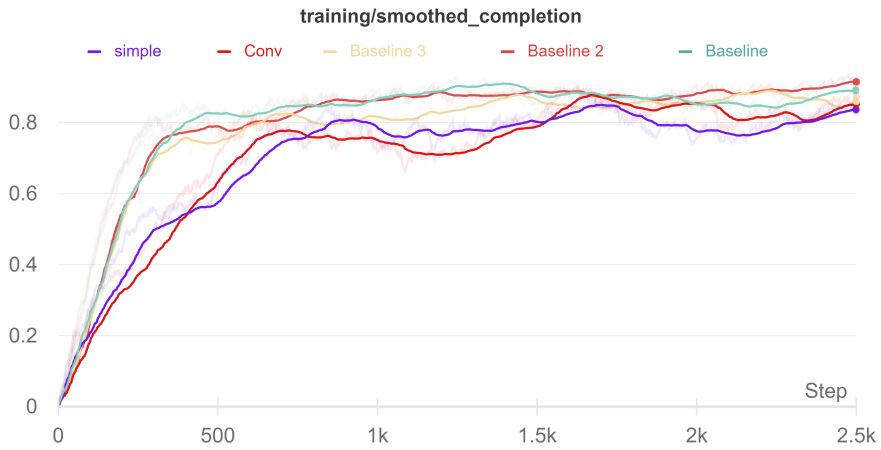

Figura A.40: Convolutional networks: completion during training episodes

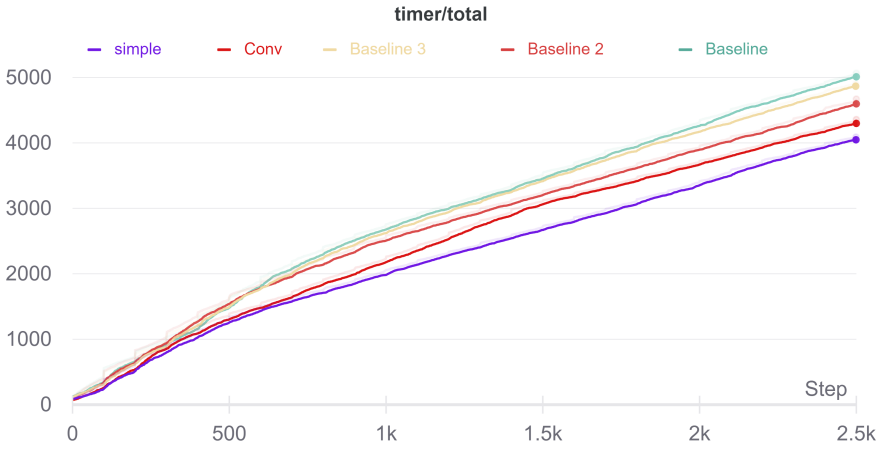

Figura A.41: Convolutional networks: Timer of the execution of the run

A.9 Deadlock

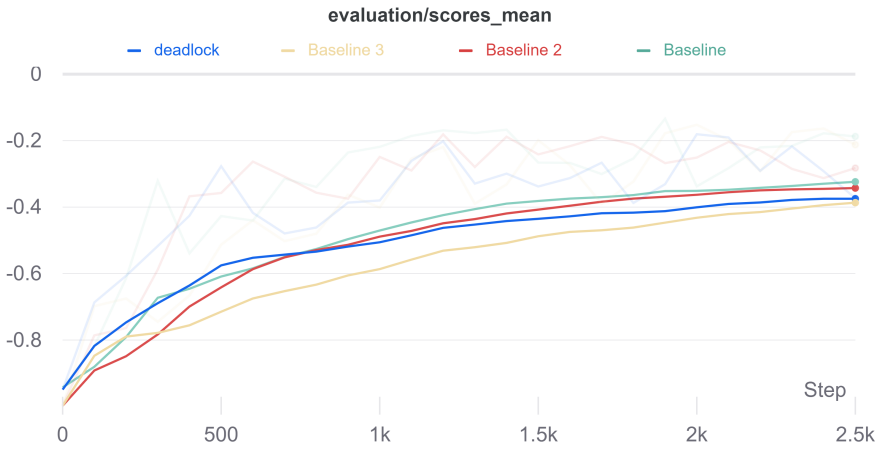

Figura A.42: Deadlock: score during evaluation episodes

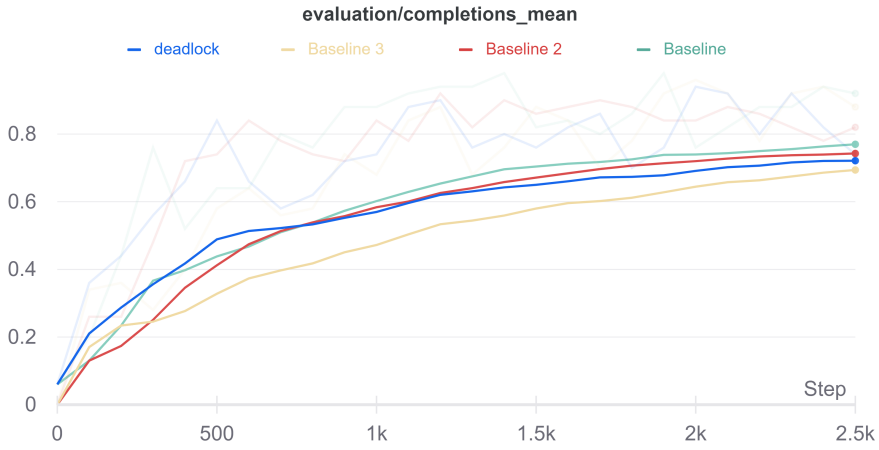

Figura A.43: Deadlock: completion during evaluation episodes

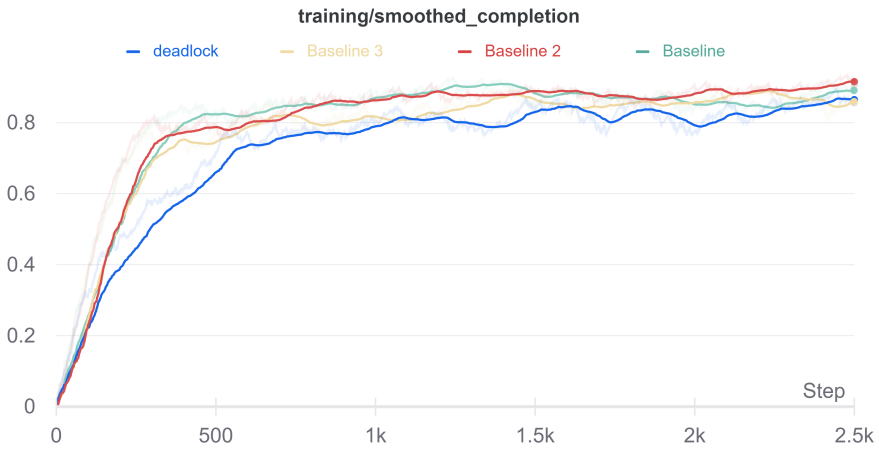

Figura A.44: Deadlock: score during training episodes

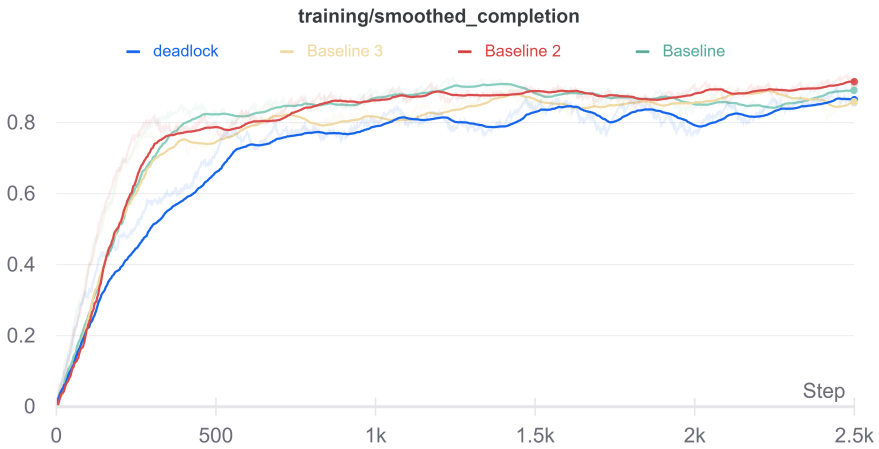

Figura A.45: Deadlock: completion during training episodes

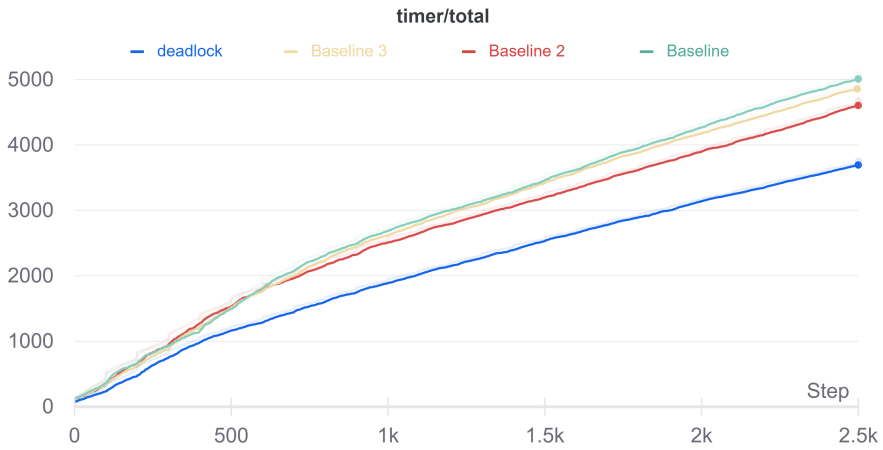

Figura A.46: Deadlock: Timer of the execution of the run

A.10 Reward: 0.1, 0.2 and 0.5 vs baseline

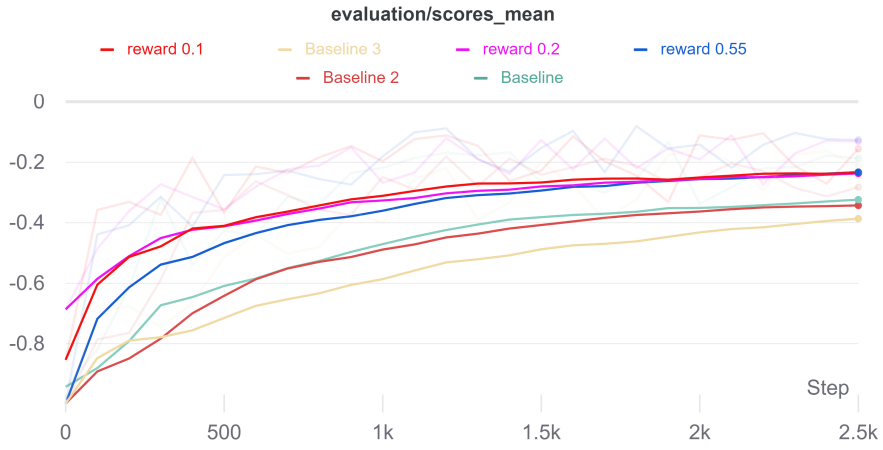

Figura A.47: Reward 0.1, 0.2 and 0.5 vs baseline: score during evaluation episodes

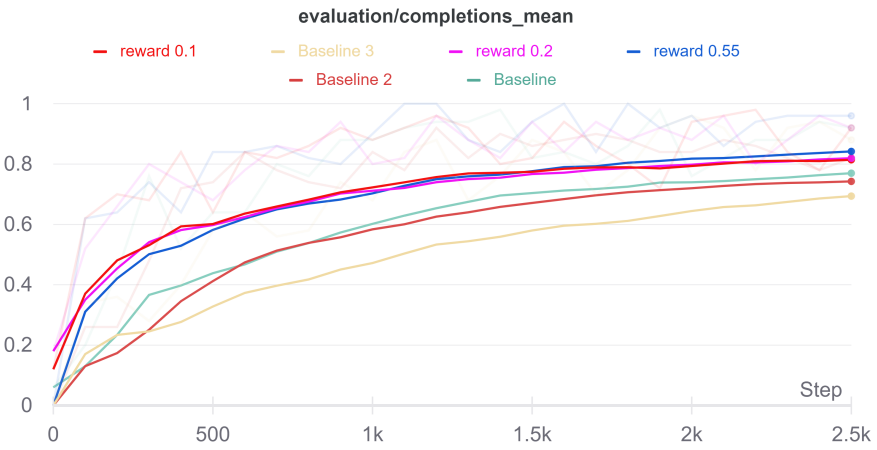

Figura A.48: Reward 0.1, 0.2 and 0.5 vs baseline: completion during evaluation episodes

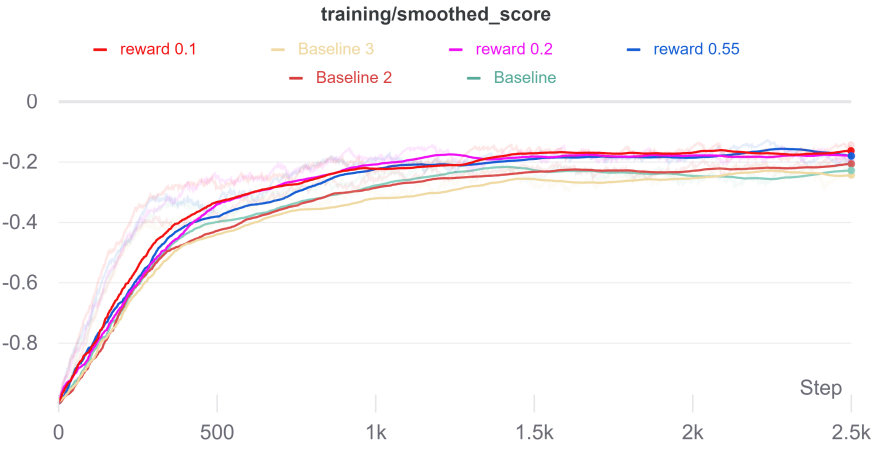

Figura A.49: Reward 0.1, 0.2 and 0.5 vs baseline: score during training episodes

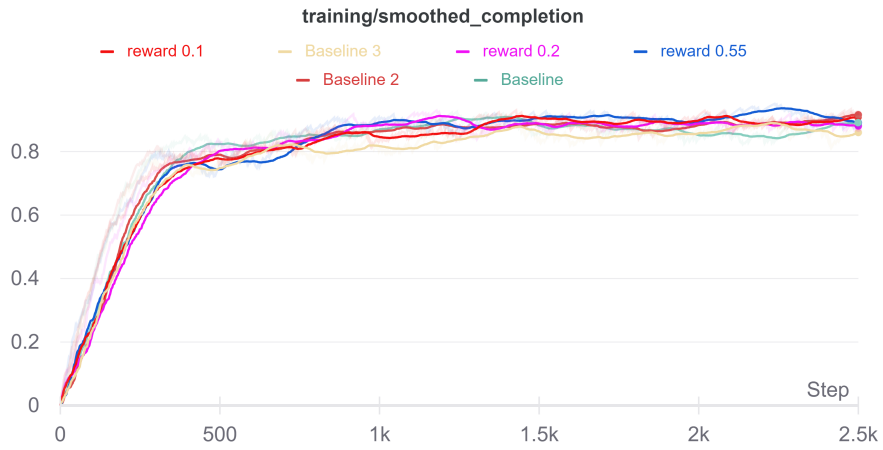

Figura A.50: Reward 0.1, 0.2 and 0.5 vs baseline: completion during training episodes

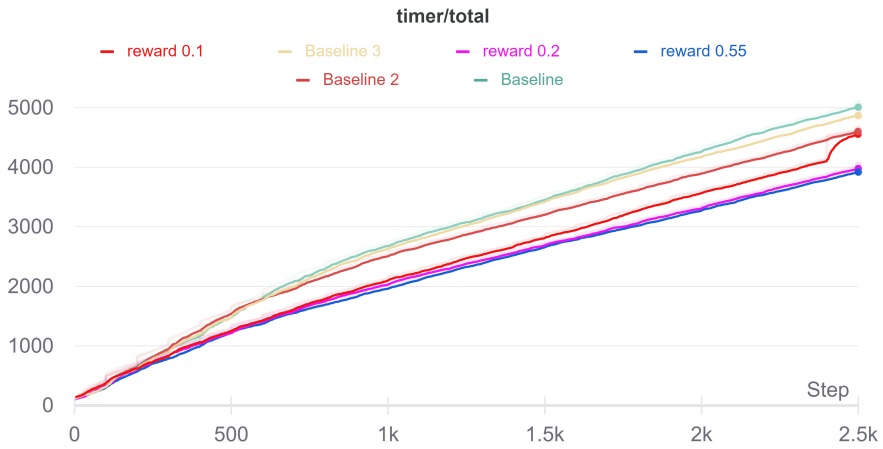

Figura A.51: Reward 0.1, 0.2 and 0.5 vs baseline: Timer of the execution of the run

A.11 Reward: 0.1 Evaluation and training environment 1 vs Evaluation and training environment 1

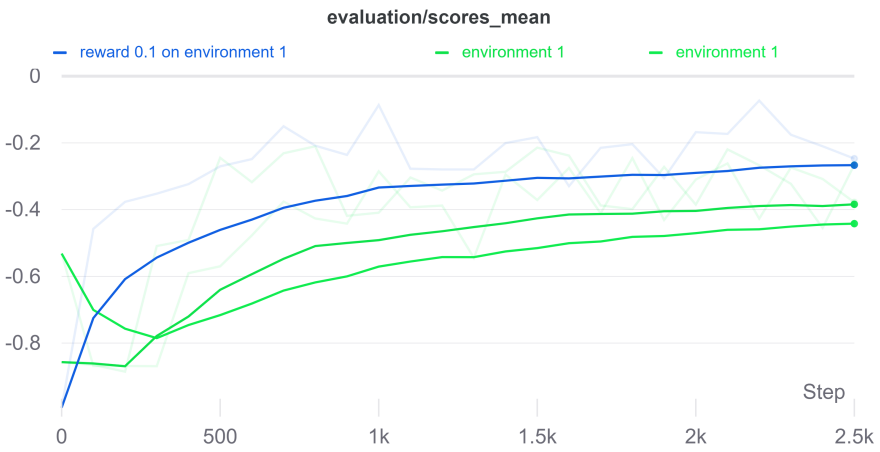

Figura A.52: Reward 0.1 e1 t1 vs e1 t1: score during evaluation episodes

Figura A.53: Reward 0.1 e1 t1 vs e1 t1: completion during evaluation episodes

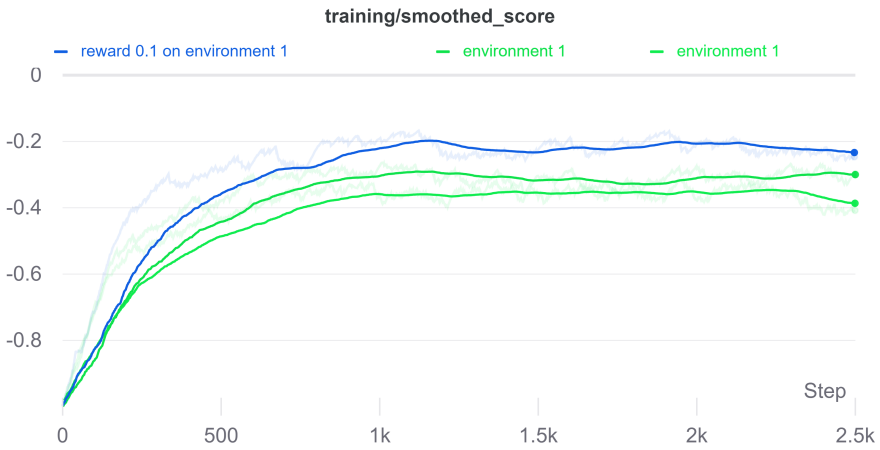

Figura A.54: Reward 0.1 e1 t1 vs e1 t1: score during training episodes

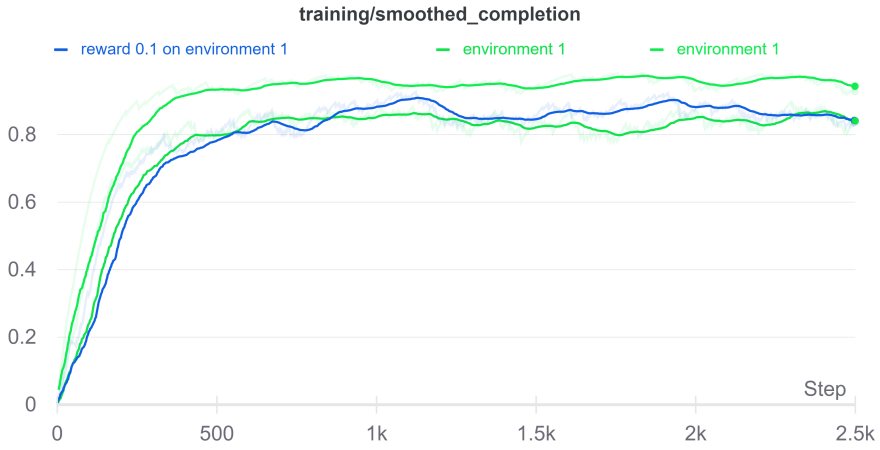

Figura A.55: Reward 0.1 e1 t1 vs e1 t1: completion during training episodes

Figura A.56: Reward 0.1 e1 t1 vs e1 t1: Timer of the execution of the run

Bibliography

Whole bibliography

[1] Alessandro Sitta Giovanni Montanari Lorenzo Sarti. «Project 5: Flatland Challenge». In: UNIVERSITY OF BOLOGNA. 2020 (cit. alle pp. 19, 21).

[2] Sharada Mohanty et al. «Flatland-RL: Multi-agent reinforcement learning on trains». In: *arXiv preprint arXiv:2012.05893* (2020) (cit. a p. 5).

[3] Hado Van Hasselt, Arthur Guez e David Silver. «Deep reinforcement learning with double q-learning». In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016 (cit. a p. 5).

[4] Jonas Wälter. «Existing and novel Approaches to the Vehicle Rescheduling Problem (VRSP)». In: University of Applied Sciences Rapperswil. 2020 (cit. a p. 23).

[5] Ziyu Wang et al. «Dueling network architectures for deep reinforcement learning». In: *International conference on machine learning*. PMLR. 2016, pp. 1995–2003 (cit. a p. 5).