Machine Learning project report

Michele Nicoletti - 1886646 - nicoletti.1886646@studenti.uniroma1.it
Lorenzo Pecorari - 1885161 - pecorari.1885161@studenti.uniroma1.it

April 2, 2025

1 Introduction

The following report aims to describe different attempts of implementing a reinforcement learning agent for efficiently solving a run of "Taxi-v3", an environment offered by the Gymnasium Documentation. The first of them uses a Tabular Q-Learning approach while the second ones can rely on the concept of neural network. They differ in general aspects, metrics and computational effort for the machine that runs each of them but with the same scope.

Each solution is developed by using Python with Pytorch, a library that allows to create tools usable by the agents. In combination with that, are used other useful libraries as Numpy and Matplotlib for data manipulation and result plotting.

For each solution, it will be provided the implementation and the results about the experiment; in particular, there will be showed, where possible, the curves of accuracy, loss and rewards obtained by the agent in raw form and, also, in smoothened average over a mobile window of 10 elements.

2 Taxi-v3, a Toy Text environment

Graphically, it consists in a 2D fancy representation of a taxi that has to pickup a passenger and drop it in a destination. More in depth, it represents a 500 discrete states grid world generated by 25 positions for the taxi, 5 for the passenger and 4 for the building/destination. In particular, each destination is represented by a color among red, green, yellow and blue and the passenger can be in one of them or inside the taxi.

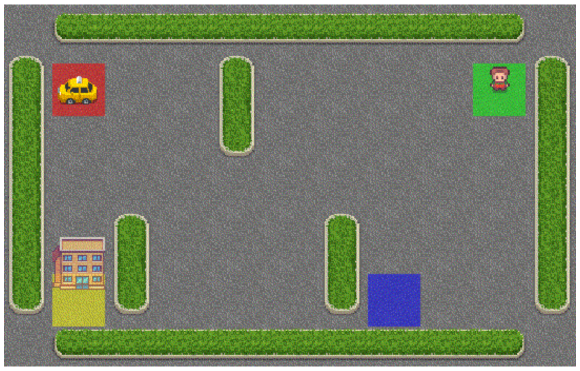


Figure 1: Graphical representation of the environment

Each time, the taxi can execute one of the 6 available moves: up, down, left, right, pick up, drop off. For a correct delivery of the passenger there will be a positive reward of 20, while an illegal pick up or drop off determines a loss of 10; for any step different from the previous actions the award will be -1.

There can be at most 300 possible initial states in which the taxi, the destination and the passenger are in different locations.

For avoiding an undefined number of actions executed during a single run, it is set by default a maximum of 200 steps per episode determining a termination state equal to "terminated" or "truncated" if the delivery happens or not.

By that, the scope of the project is to train a model capable to correctly solve the environment while maximizing the total reward.

The environment is imported by the library `gymnasium` through the method `gymnasium.make("Taxi-v3")`. For more details, the official page of the environment is available here: `https://gymnasium.farama.org/environments/toy_text/taxi/`

3 First solution: Tabular Q-Learning

3.1 Idea behind the solution

The following implementation uses a table where values are stored, accessed and updated each time an action is taken and executed. More in detail, using the concepts of Q-values and "epsilon-greedy" approach, the agent is able to choice the better action in each state it will be by accessing the correct element of the table.

The structure of the table is characterized by a 500×6 matrix where each row is related to a possible state of the agent while each column is the relative action among the possible ones. Each element of the structure contains a value approximated to the predicted reward the agent will get if in such state it executes the action of the corresponding column.

Initially, each value of the matrix is set to 0 but, with the update method, it will be gradually populated by different values obtained by the formula of the Q-value:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (r + \gamma \cdot \underset{a' \in A}{max}(Q(s',a')) - Q(s,a))$$

In detail, the action that allows to achieve the maximum reward after the execution of an action leading to a new state contributes to update the element of the table corresponding to the actual state. Thanks to these updates, the agent will be able to have an estimation of the reward of an action done in a specific state just by looking at the table.

3.2 Implementation

This solution needs of two different classes: one for handling the table and one for the agent.

The class `QTable` is the matrix representing the table and it is initialized through `np.zeros((states, actions))`, where `states` and `actions` are referred to the rows and columns of it. The update is performed by `QTable.update` that uses the formula previously showed.

The class `Agent` generates the objects in charge to solve the environment. It uses the table through `QTable` and also stores other parameters as α, ϵ and γ. The main functions are:

- `choose_action`, which picks a random float and if it is lower than ϵ it samples an action otherwise it chooses the action returning the highest reward given the current state;
- `update_table`, that gets the current Q-value, the new one by updating the table and evaluates the loss as the squared difference between the two values;
- `train`, where for each episode it resets the environment and, until it is not done, it chooses and executes the action by the current state, tracking the relevant information as the total reward, the average loss and the way the episode ends.

The agent enjoys of a good grade of exploration during the execution thanks to the "ϵ-greedy" approach. In detail, at each episode an action is randomly chosen with probability ϵ instead of on the basis of the maximum reward it can return. At each episode, the value of ϵ becomes $\epsilon = \epsilon \cdot \epsilon_{dec}$

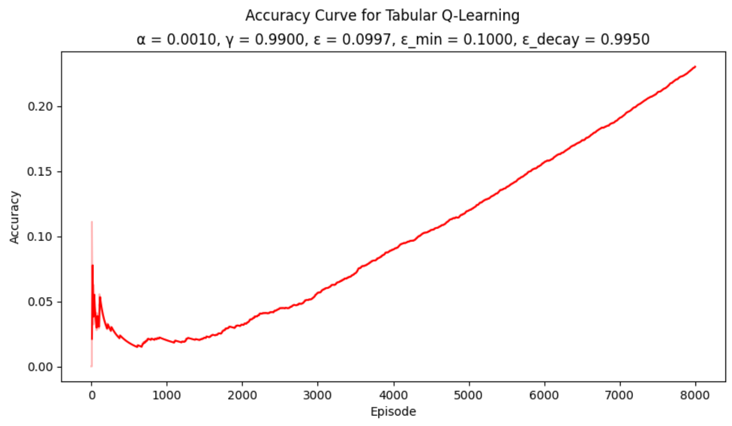
until it does not reach the minimum value ϵ_{min}, where ϵ_{dec} is the decay factor. It is important to define $\epsilon_{dec} < 1$ for avoiding an increasing value of ϵ over the episodes.

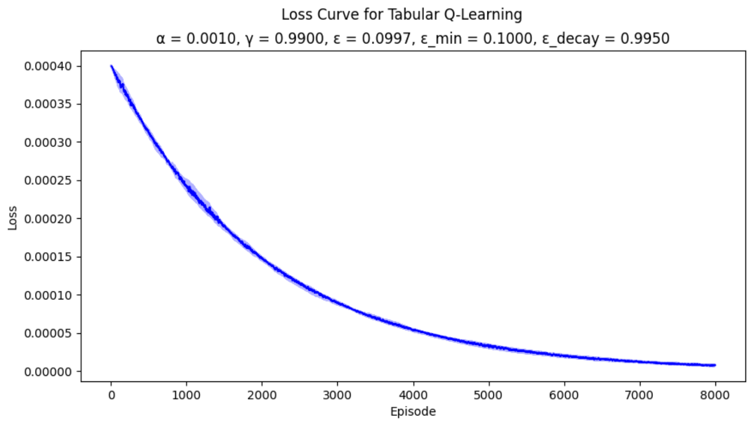
Once the execution of the function `train` terminates, it is possible to plot the metrics tracked over the episodes.

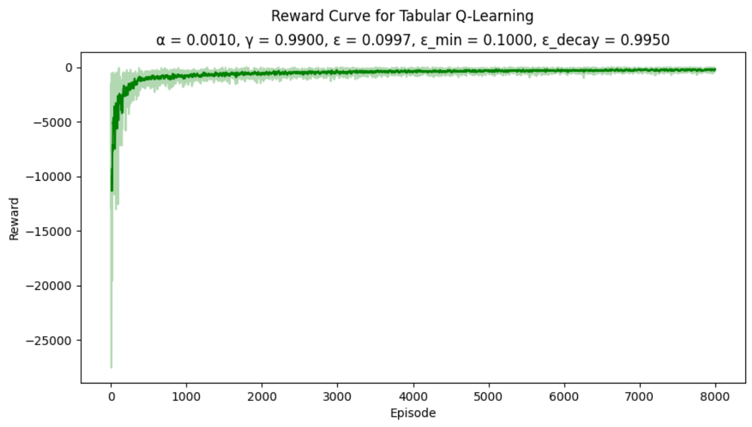
3.3 Experiments

In order to perform an analysis over the performances, it was needed to tune the parameters of the agent. More in depth, there were changes during the executions, the learning rate α, the decay factor of ϵ and the number of episodes for have a different view of the curves.

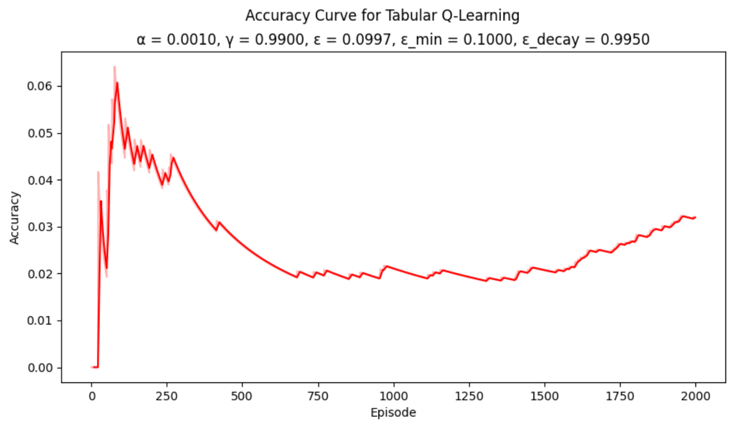
3.3.1 $\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{decay} = 0.995$, episodes = 8000

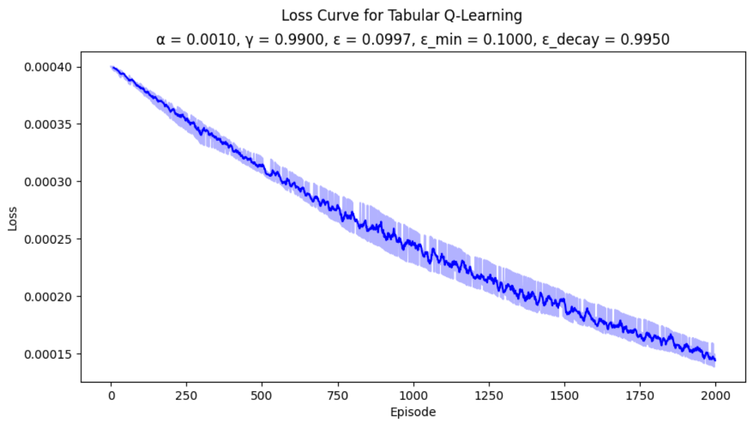


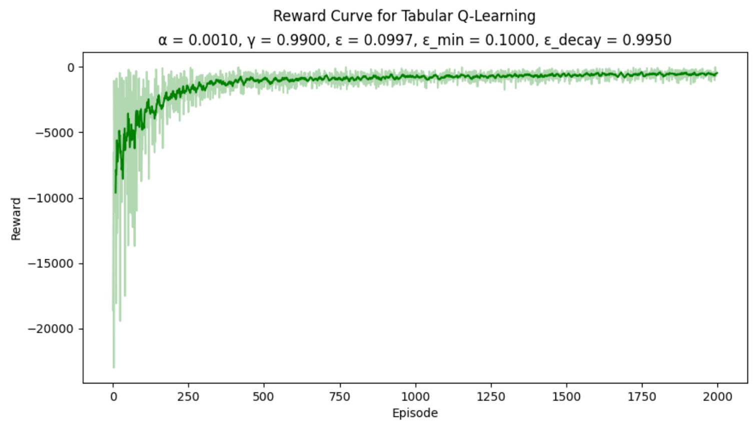




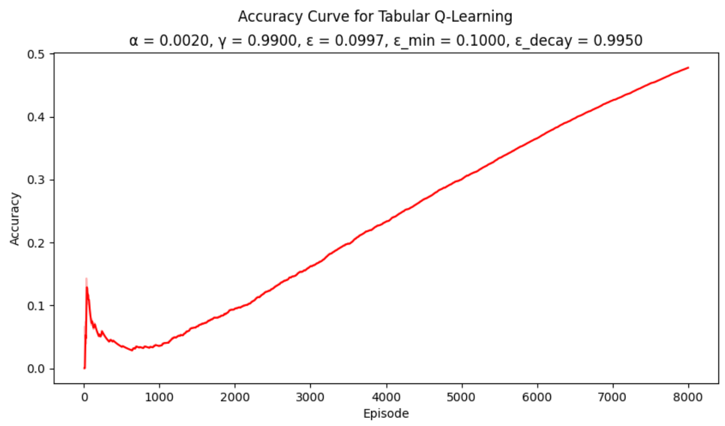
3.3.2 $\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{decay} = 0.995$, episodes $= 2000$

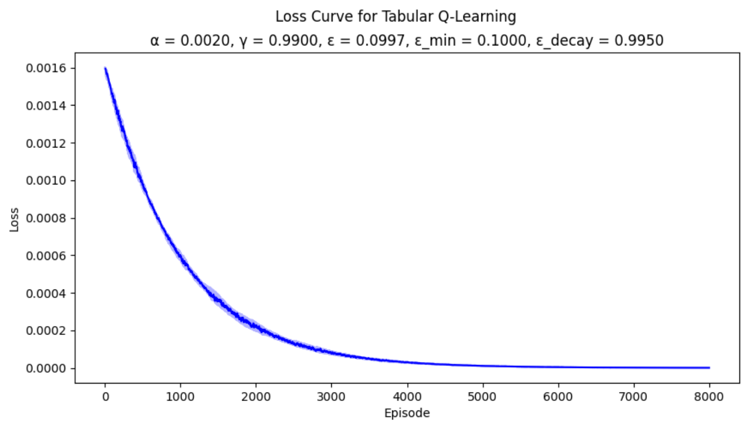


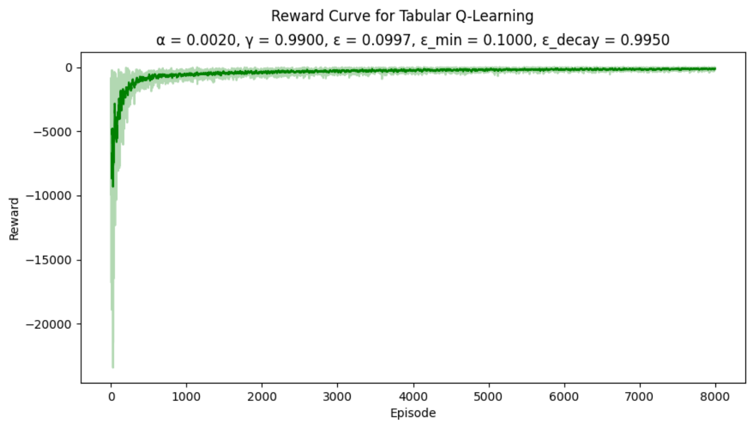




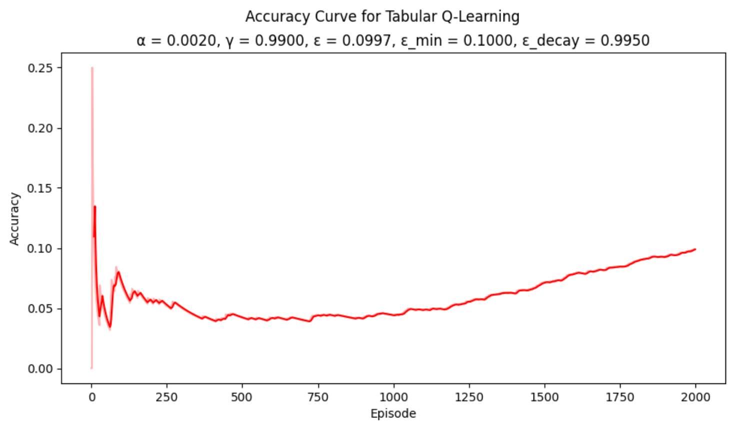
3.3.3 $\alpha = 0.002$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{decay} = 0.995$, episodes = 8000

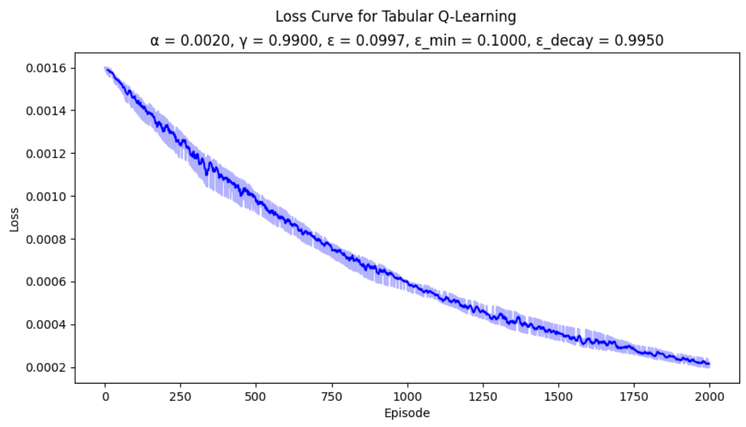


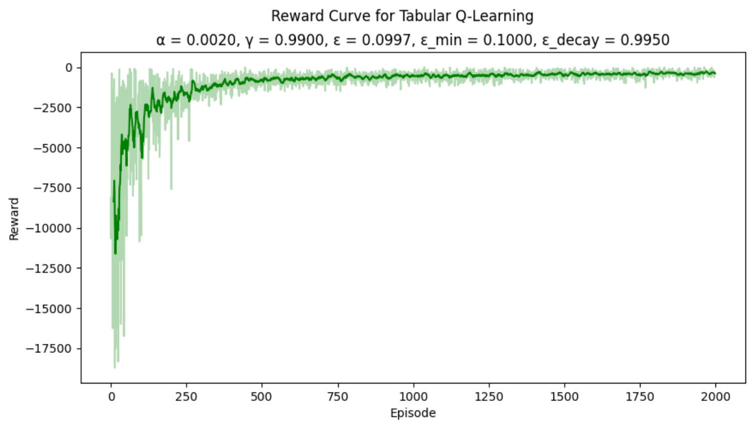




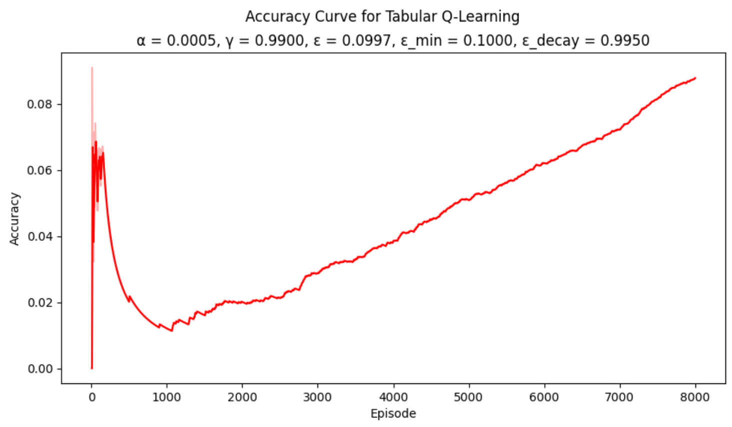
3.3.4 $\alpha = 0.002$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{decay} = 0.995$, episodes = 2000

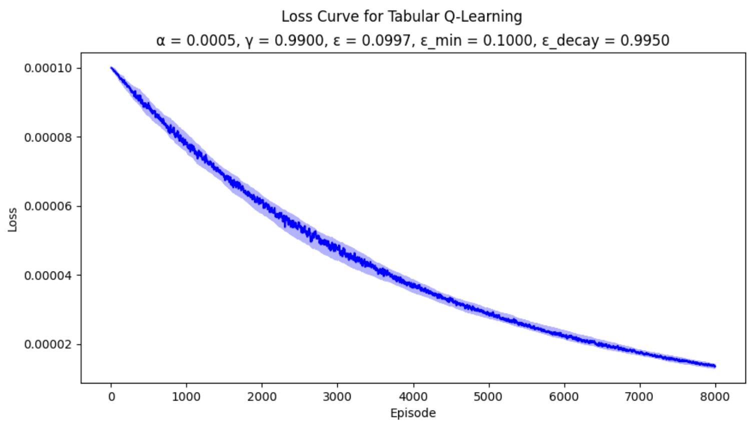


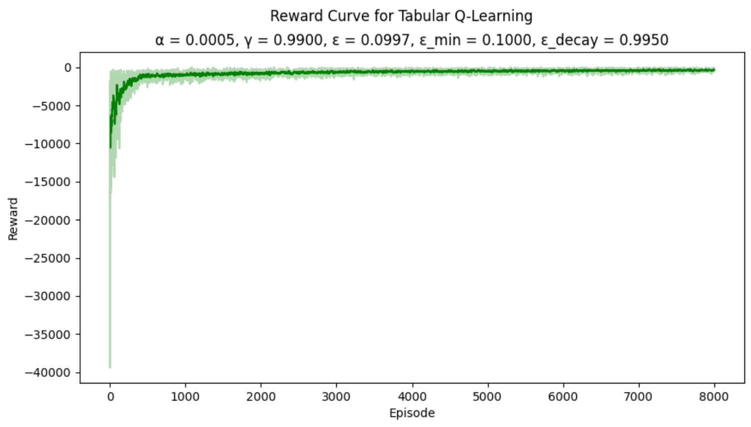




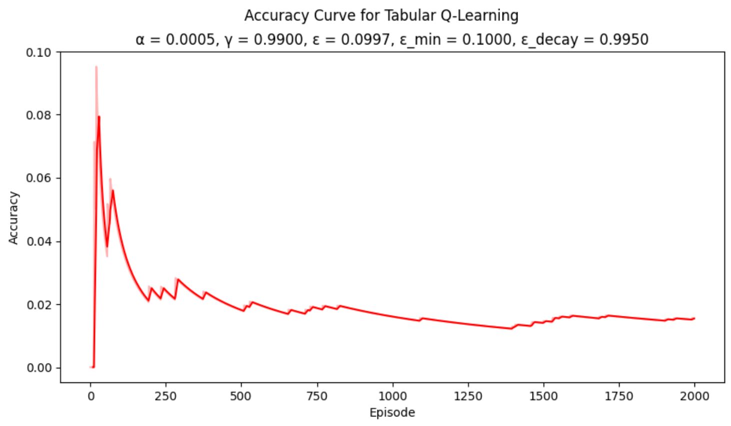
3.3.5 $\alpha = 0.0005$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{decay} = 0.995$, episodes $= 8000$

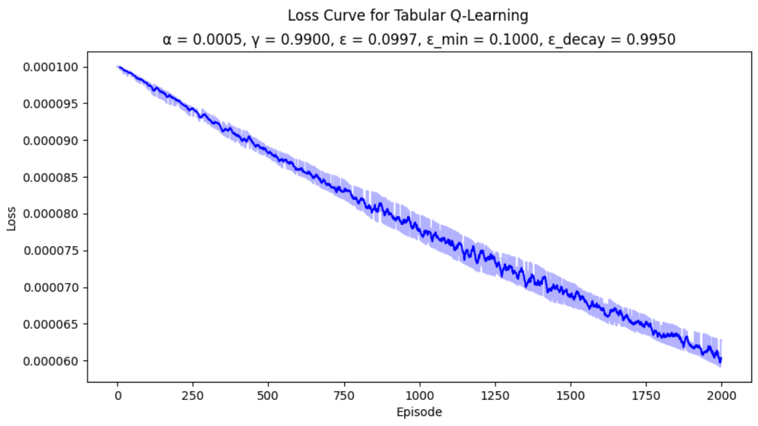


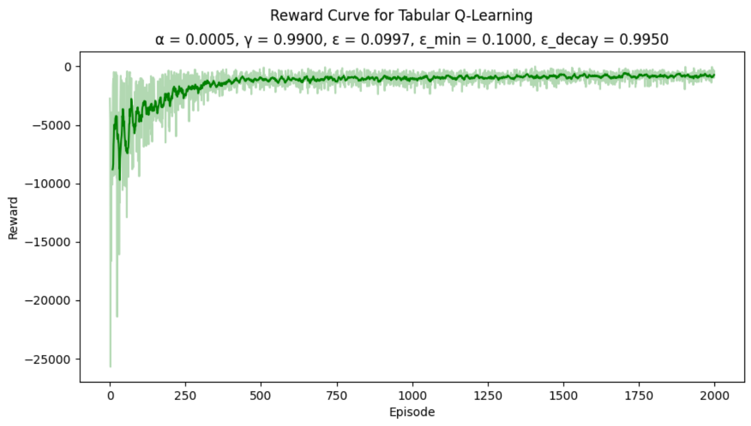




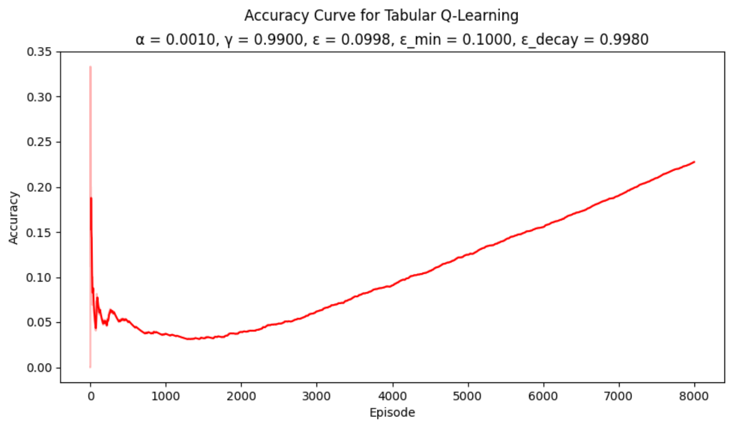
3.3.6 $\alpha = 0.0005$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{decay} = 0.995$, episodes $= 2000$

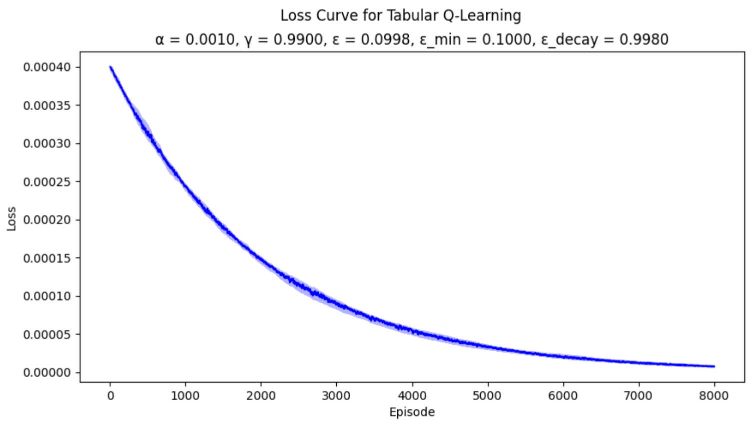


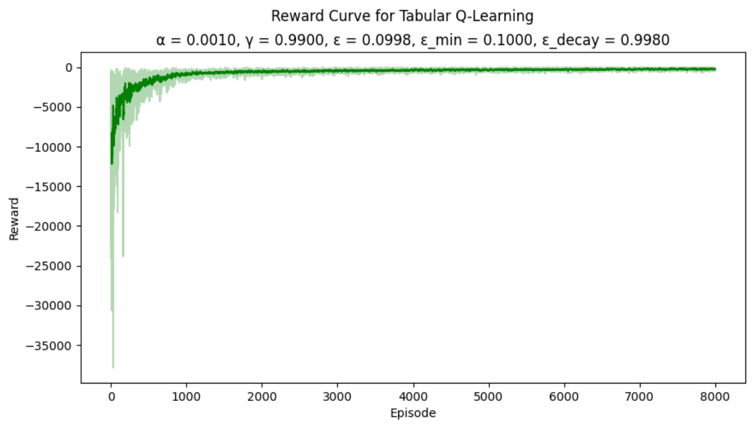




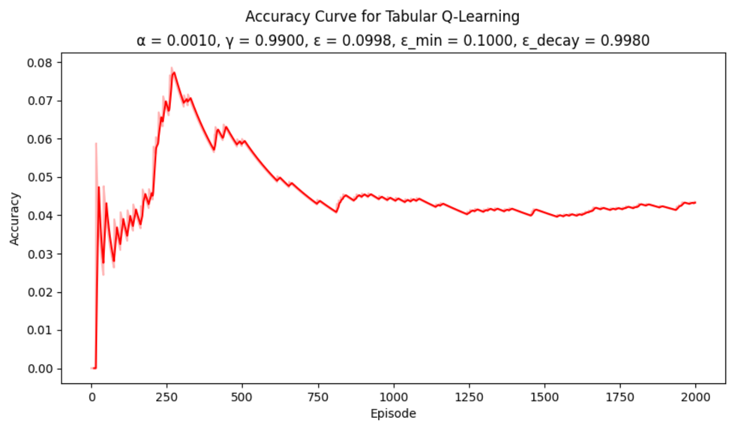
3.3.7 $\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{decay} = 0.998$, episodes = 8000

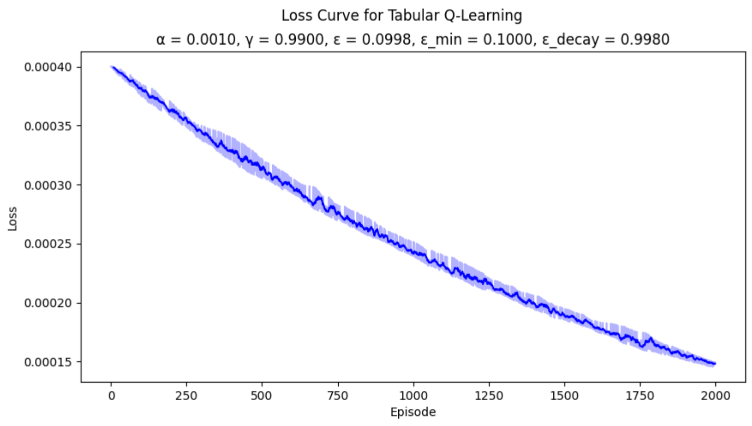


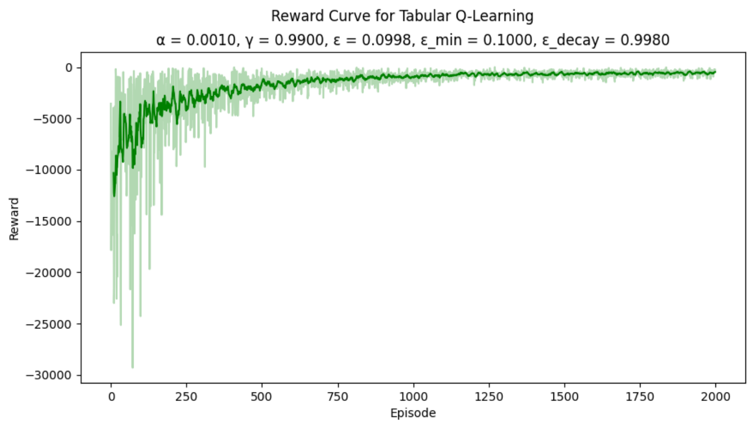




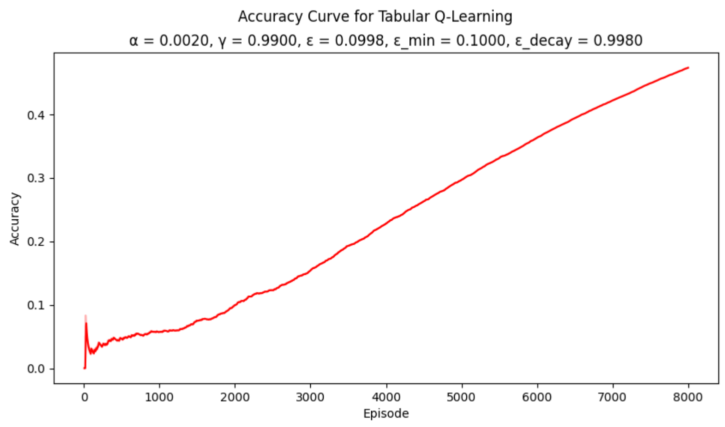
3.3.8 $\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{decay} = 0.998$, episodes = 2000

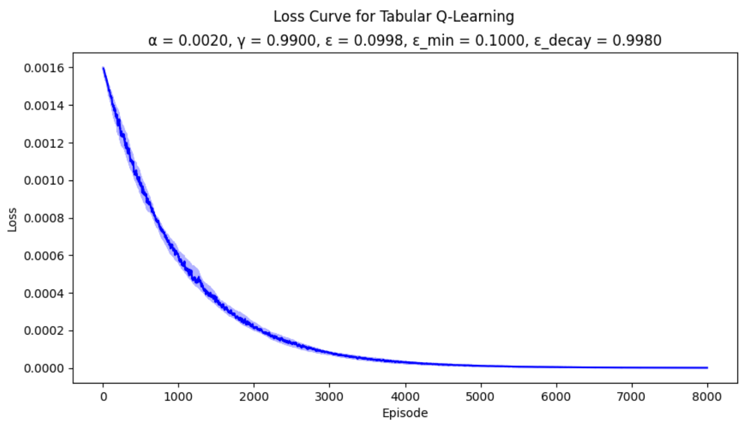


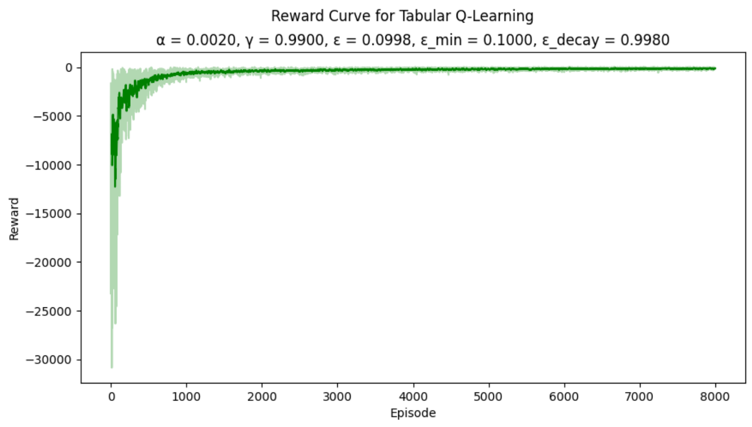




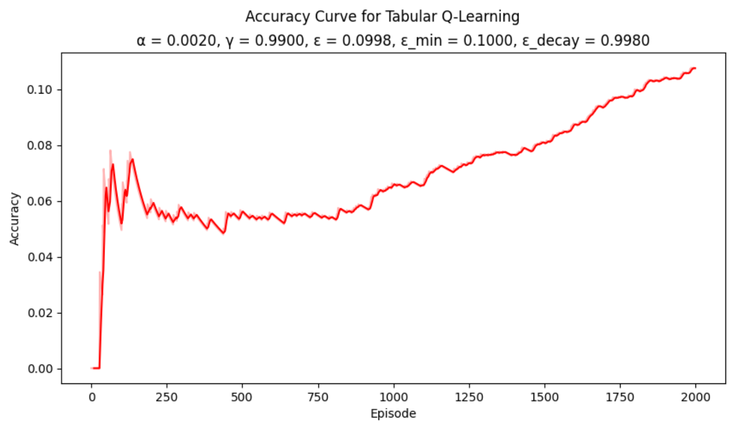
3.3.9 $\alpha = 0.002$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{decay} = 0.998$, episodes = 8000

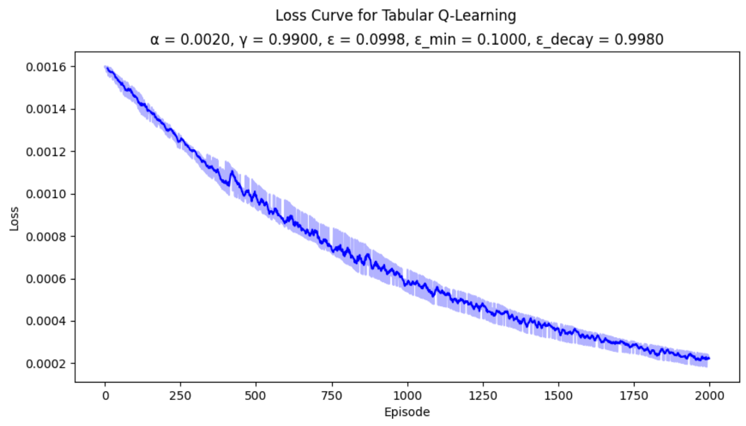


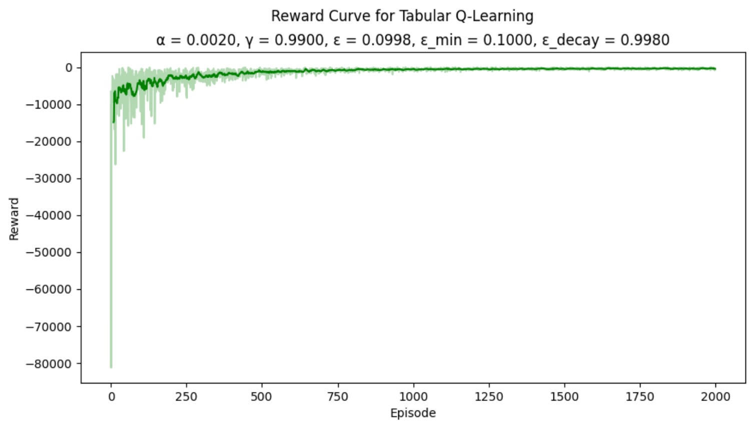




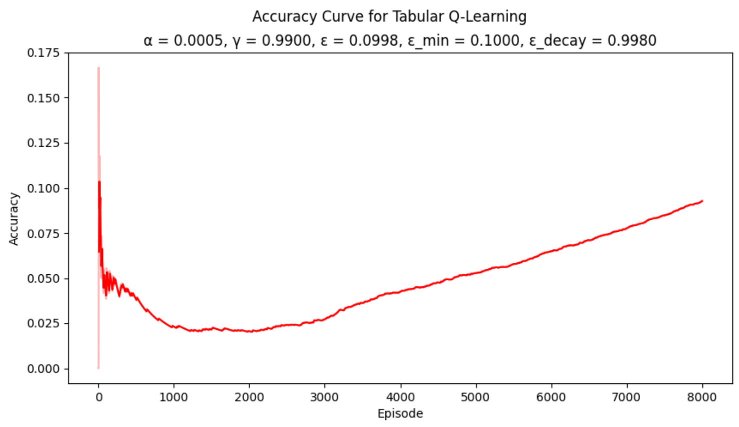
3.3.10 $\alpha = 0.002$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{decay} = 0.998$, episodes = 2000

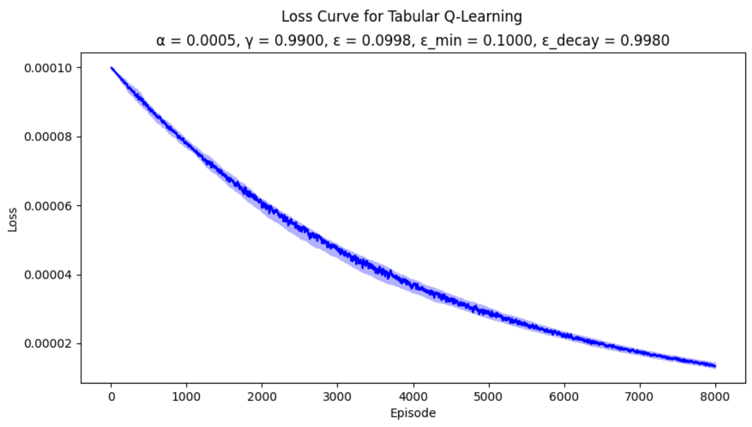


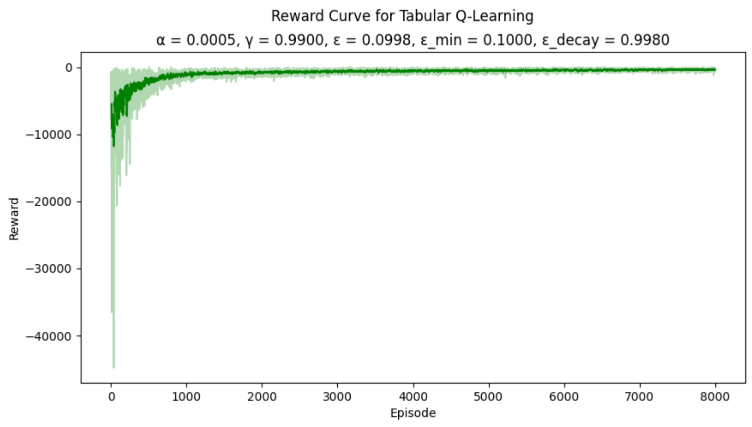




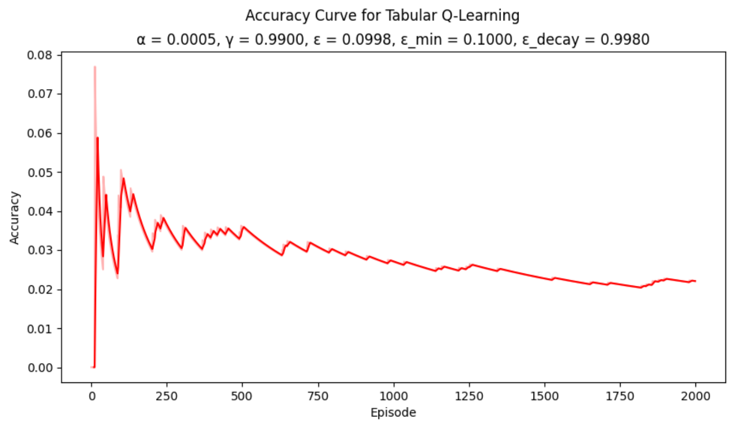
3.3.11 $\alpha = 0.0005$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{decay} = 0.998$, episodes $= 8000$

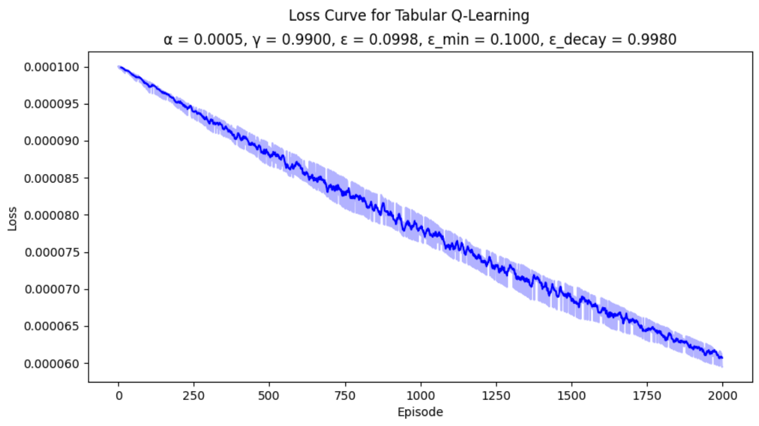


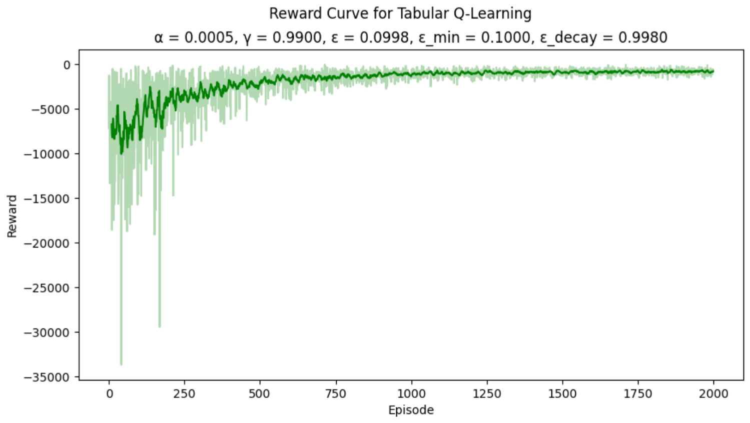




3.3.12 $\alpha = 0.0005$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{decay} = 0.998$, episodes = 2000







3.4 Observations

3.4.1 Fixed decay factor $\epsilon_{dec} = 0.995$, variable learning rate α

From a starting points where $\alpha = 0.001$ and $\epsilon_{dec} = 0.995$ (3.3.1), the experiments show that it is possible to evaluate a convergence to approximately stable values of rewards from the 500^{th} episode. This can be better analyzed looking at the plots referred to the run with 2000 episodes (3.3.2), even if the total reward barely reaches the value 0. This behavior can be checked also by looking at the accuracy plots, with an ascending trajectory of the curve in the case of 8000 episodes; less visible is with the 2000 episodes situation but also with a weak growth of the value. With the gradual update of the table, the loss between the actual and the new Q-value starts to descend in both the plots.

By setting $\alpha = 0.002$ (3.3.3 and 3.3.4), there is quite clear improvement of convergence speed to more stable values with that event around the episode number 300; about the accuracy, between the 500^{th} and the 750^{th} can be observed a stable behavior and after the 1000^{th} episode starts to grow. The loss rapidly tends to reach values around 0.

About the case where $\alpha = 0.0005$ (3.3.5 and 3.3.6), there is a similar trend about rewards as the previous tests, with a more linear descending of the loss and a worse and constant course of the accuracy.

3.4.2 Fixed decay factor $\epsilon_{dec} = 0.998$, variable learning rate α

An initial analysis on the tests conducted on $\alpha = 0.001$ (3.3.7 and 3.3.8), leads to evaluate a slower and regular growth of the rewards curve between 750^{th} and 1250^{th} episodes with a more stable behavior after the 1250^{th} episode with a propensity to the value 0. This can be also seen in the accuracy plot, where a similar trend is already observed in the rewards curve. Similarly to previous and different experiments, the loss curve slowly tends to reach low values.

Differently from a lower learning rate, with $\alpha = 0.002$ (3.3.9 and 3.3.10) the rewards curve seems to be way faster to converge to quite good values tending to 0. Accordingly with that observation, the accuracy shows a rapid (and linear-similar) growth to values near to 0.5 in the case of 8000 episodes plots (3.3.9) and, in the "zoomed-in" plots with 2000 episodes, later than the 2000^{th} episode already reaches 0.2 .

Trying to read the plots referred to $\alpha = 0.0005$ (3.3.11 and 3.3.12), the agent shows a good growth of accuracy starting from the 3000^{th}; a marked difference with respect to the previous results is coming inside the loss curve, where there is a slower decrease of the value with the continuation of the execution. The rewards curve, in both 8000 episodes and 2000 episodes plots, outlines a similar behavior with higher values of learning rate.

4 Second solution: Neural Network

4.1 Idea behind the solution

As a more complex computational model for artificial learning, the neural network is based on a similar architecture as the human brain. In particular, it is possible to define nodes as "neurons" placed in so called "layers". In this way, each neuron works as a mathematical function that receives an input and generates an output that can be given as input for other nodes. In its basic configuration, there will be an input and an output layer; for improving performances it might be needed to add one or more "hidden" layers, in charge of elaborating information in order to achieve the learning.

For storing weights and information about computation and past experiences, it is required a `Replay Buffer`: it consists on a data structure like a buffer where is possible to push elements and pick some of them randomly.

The agent works directly with a neural network to elaborate data and a replay buffer for storing them.

4.2 Implementation

In this solution there are some classes: `Agent` for the agent, `ReplayBuffer` for the replay buffer and three different ones for neural networks.

In particular, it has been defined a class `Lx_QNet` that implements a network with an input layer, $x-2$ hidden layers and an output layer. The parameters for the class are `state_dim` and `action_dim`, representing the dimension of the state returned by the environment and the size of the structure containing the possible actions executable. Basing on the previous class, the three different neural networks are developed with `L3_QNet`, `L4_QNet` and `L5_QNet` respectively for 1, 2 and 3 hidden layer other than input and output ones. The main core of these classes are the functions `nn.Linear` and `torch.relu` directly taken from the library Pytorch and its relative module for neural netowrks. They operates for generating a linear layer with specific input and output size, and for defining which criteria is used for taking decisions (following $g(\alpha) = max(0, \alpha)$). In detail, `nn.Linear` is used when the network is initialized while `torch.relu` is called by the forward function of the class. The flow of the data through the network follows a sequential behavior for giving an output by the previously given input: this operation is performed by the `Lx_QNet.forward` function.

About `ReplayBuffer`, the implementation follows few functions for pushing elements and randomly taking a batch (subset) of elements, with an initialization that sets the size of the buffer by a given parameter.

The class `Agent` is mainly characterized by the following functions:

- `update_network`, it loads the actual state of the network by using the functions `load_state_dict` and `state_dict` taken from Pytorch;
- `select_action`, as for the tabular it uses a "ϵ-greedy" approach for choosing the next action

to perform, where with probability $1-\epsilon$ is taken an action by reasoning on the state in tensor form (obtained by `torch.tensor` and `torch.nn.functional.one_hot` for ensuring uniqueness);

- `replay`, it loads the data from a batch of size 64, computes the tensorization on each parameter and on the current and next states performs also the one-hot; it gives these two states to the network for obtaining the Q values and the loss that is used for calculating the gradients through the backward function.

The agent, when possible, by the function `to(self.device)` loads directly the model into the dedicated graphical processing unit for better performances. Another technical adoption for improving metrics is the optimizer "Adam", that uses a mobile window of gradients and squared gradients in order to change automatically the learning rate for each parameter.

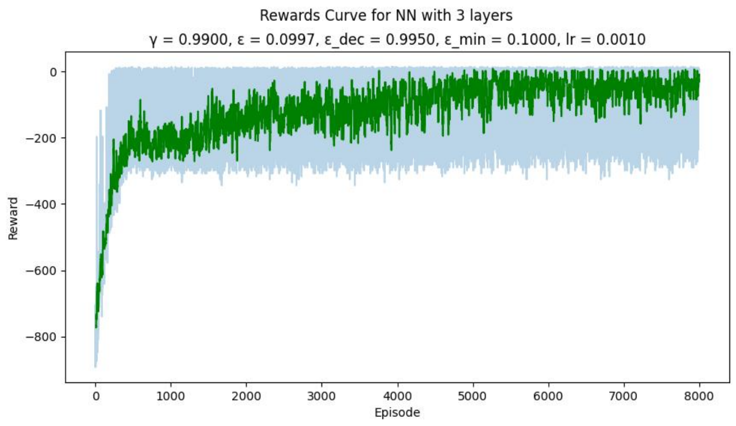
The `Train` function is the one that performs all the computations. It takes as input the number of episodes, γ, ϵ, α, ϵ_{dec}, ϵ_{min}. First of all, it creates the three different agents having the neural networks with 3, 4 and 5 total layers respectively. For each of them and for each episode, as done for the tabular solution, it resets the environment and, while the episode is not done, chooses the action by the current state and it is executed. Later that, the relevant information returned by the environment are pushed into the replay buffer, and given to the network by the `replay` function. The value ϵ is updated at the end of the episode. Each time an agent terminates its execution, the plots referred to accuracy, loss and rewards will be printed in their dedicated files.

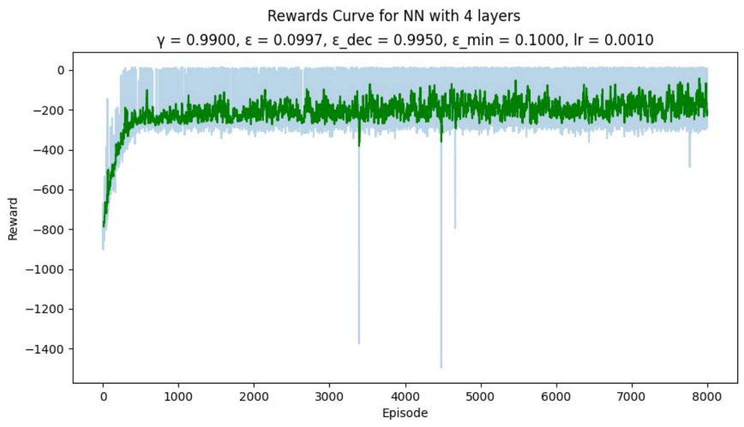
4.3 Experiments

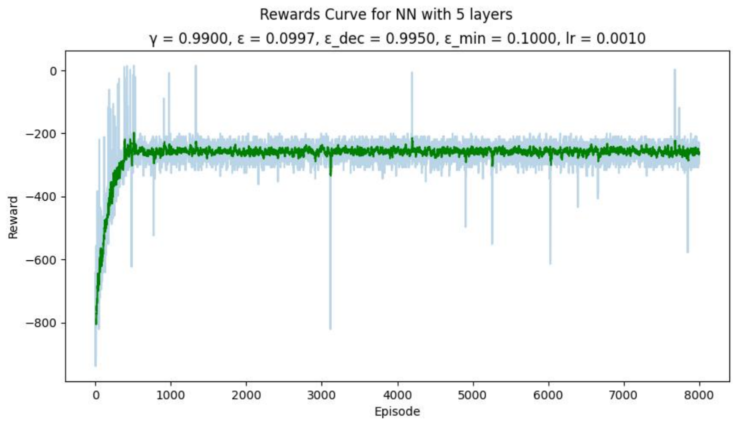
The tests done on this solution are mainly focused on the change of α, ϵ_dec and width of hidden layers of the networks. Not all of them are reported but only the most interesting and their relative rewards plots. For reason of complexity, the number of episodes per test is fixed to 8000.

4.3.1 Networks with 16 neurons for each hidden layer

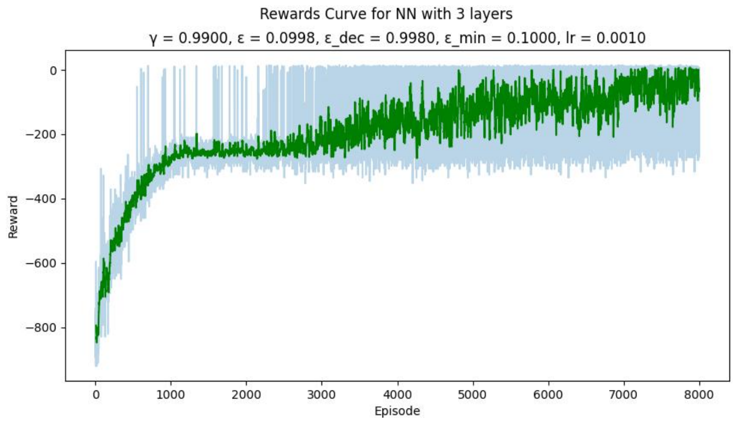
$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.995$

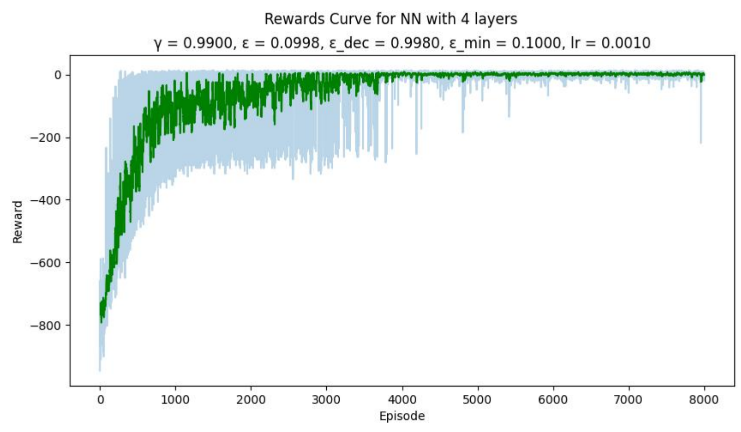


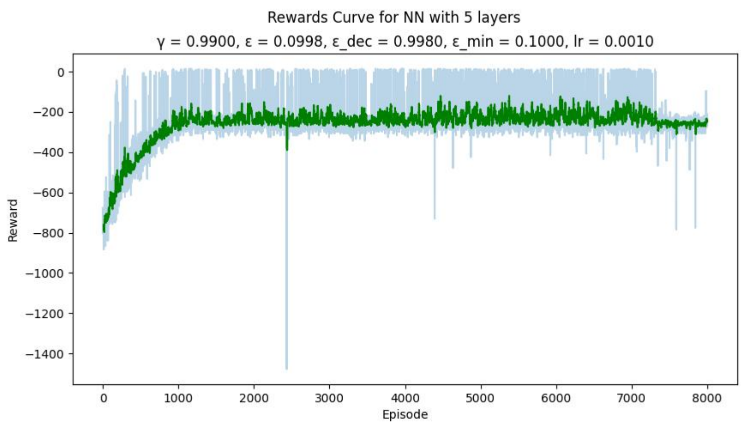




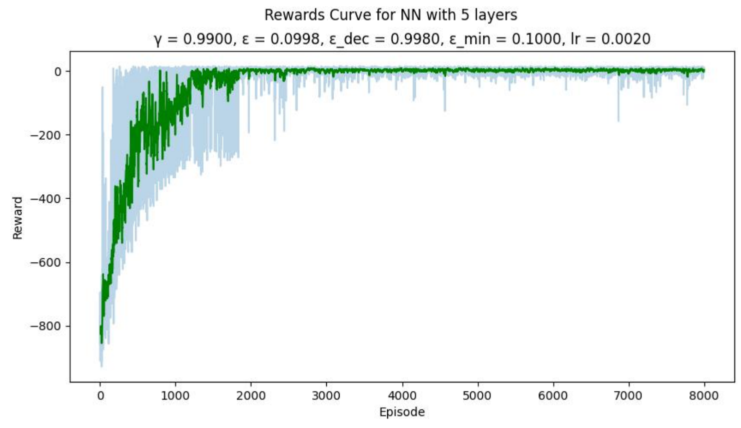
$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.998$







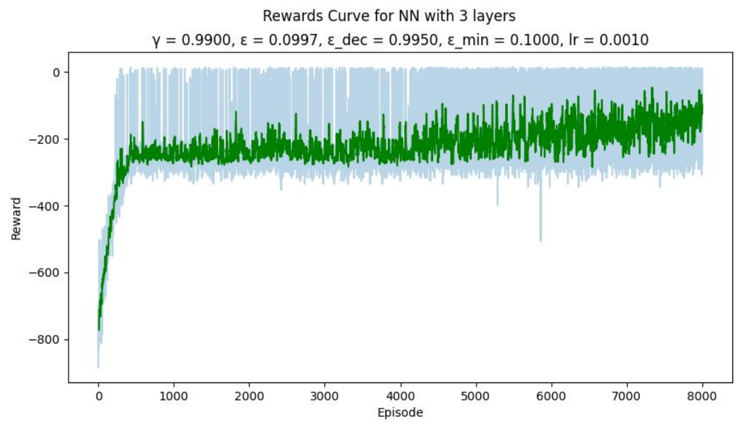
$\alpha = 0.002$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.998$

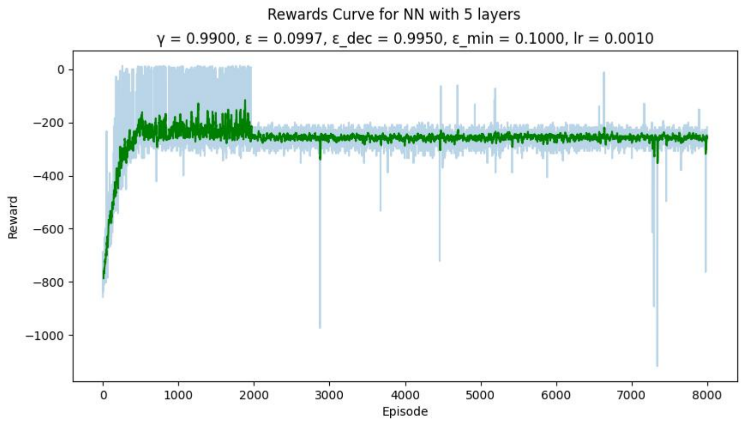


Observation on the test With few neurons for each hidden layer, the results seems to be not optimal or they tends to be good only after a lot of episodes. The only network that shows a lack of performances in the first two cases is the one with 5 layers but, with an increase of the learning rate and a relative more greedy approach on the learning, achieves better results.

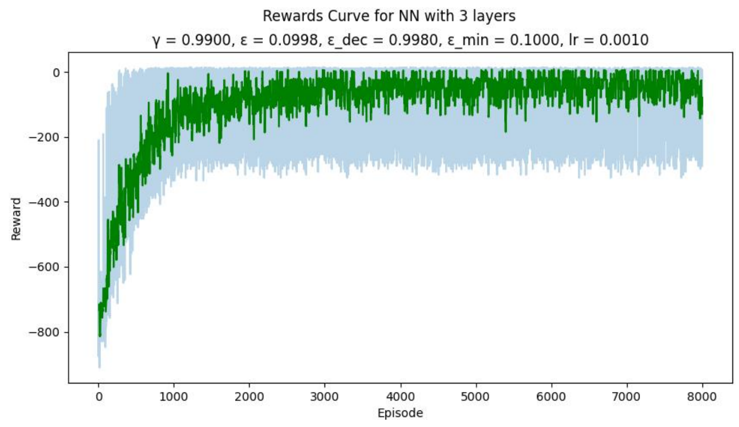
4.3.2 Networks with 24 neurons for each hidden layer

$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.995$

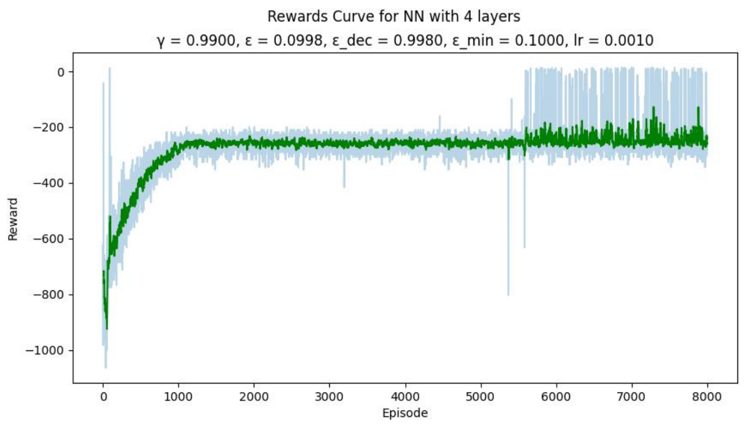




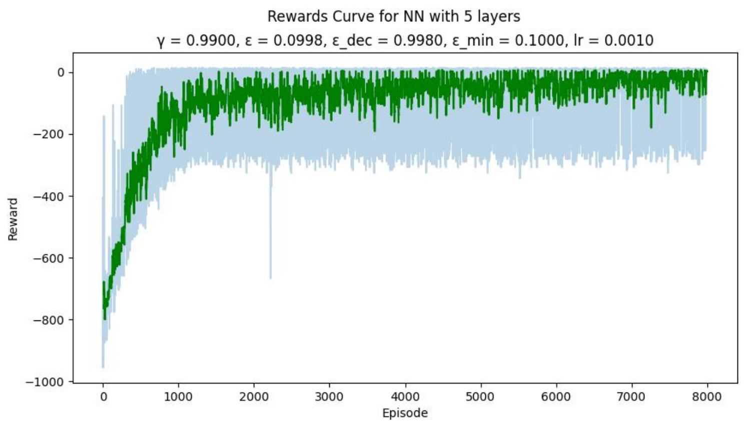
$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.995$



Rewards Curve for NN with 3 layers
γ = 0.9900, ε = 0.0998, ε_dec = 0.9980, ε_min = 0.1000, lr = 0.0010
Reward
Episode
0
−200
−400
−600
−800
0
1000
2000
3000
4000
5000
6000
7000
8000

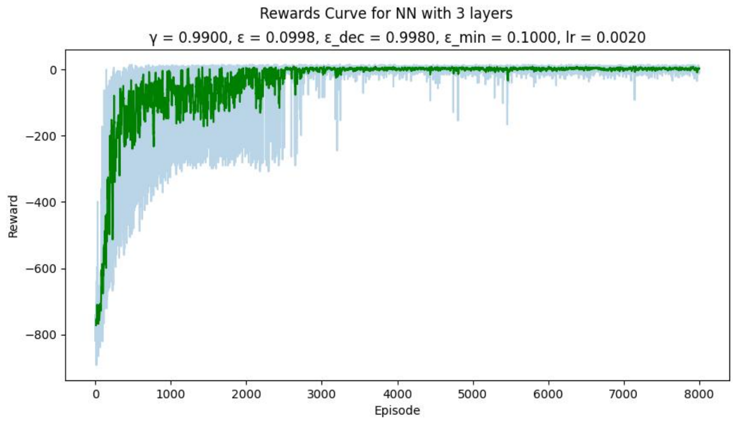


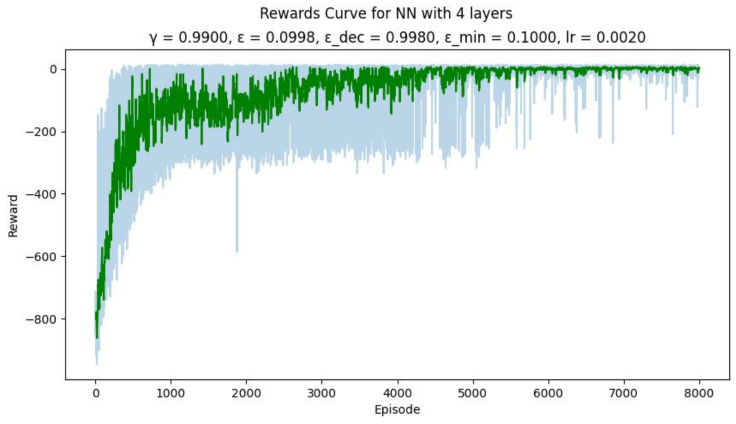
Rewards Curve for NN with 4 layers
γ = 0.9900, ε = 0.0998, ε_dec = 0.9980, ε_min = 0.1000, lr = 0.0010
Reward
Episode
0
−200
−400
−600
−800
−1000
0
1000
2000
3000
4000
5000
6000
7000
8000



Rewards Curve for NN with 5 layers
γ = 0.9900, ε = 0.0998, ε_dec = 0.9980, ε_min = 0.1000, lr = 0.0010
Reward
Episode
0
−200
−400
−600
−800
−1000
0
1000
2000
3000
4000
5000
6000
7000
8000

$\boldsymbol{\alpha = 0.002,\ \gamma = 0.99,\ \epsilon = 1.0,\ \epsilon_{min} = 0.1,\ \epsilon_{dec} = 0.998}$ [H!]

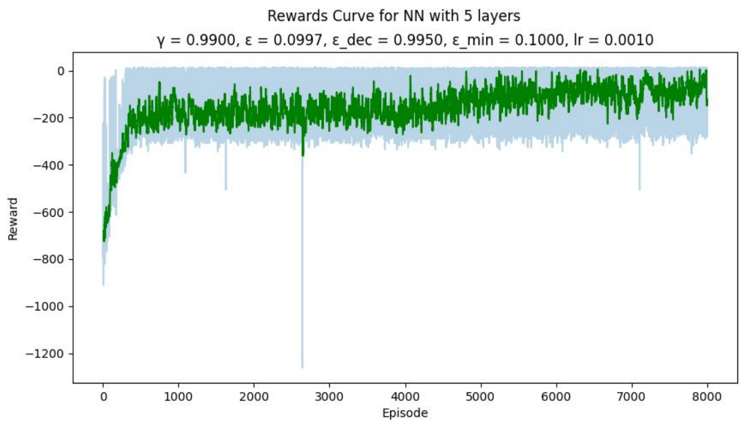




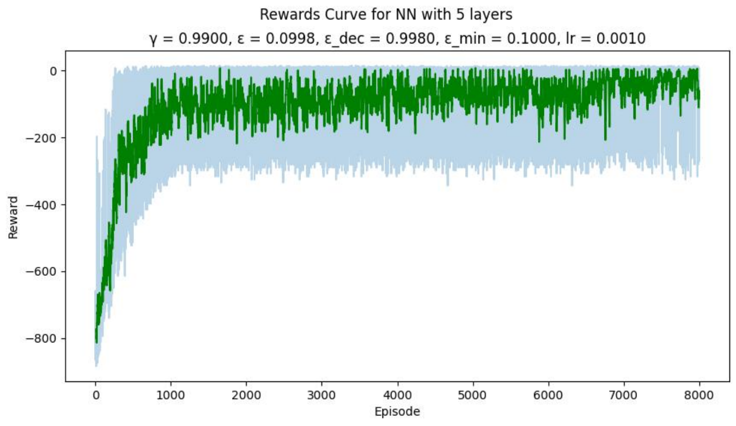
Observation on the test Adding some neurons for each layer allows some slight improvements even if the performances seems to be still poor. Operating with a tune of the parameters it is anyway possible to have better rewards without changing radically the network used by the agents.

4.3.3 Networks with 32 neurons for each hidden layer

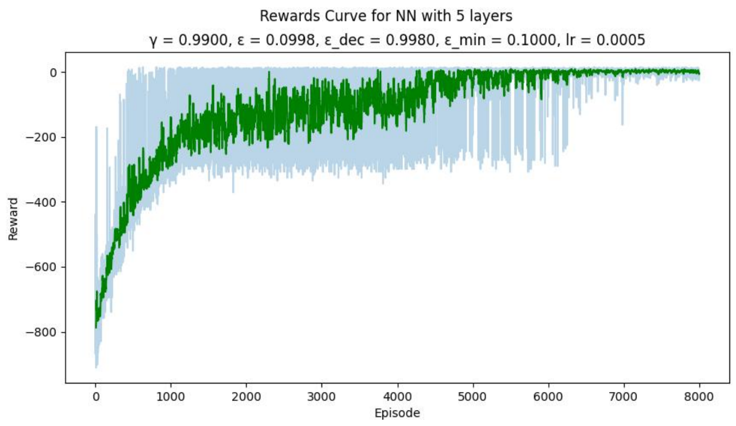
$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.995$



$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.995$



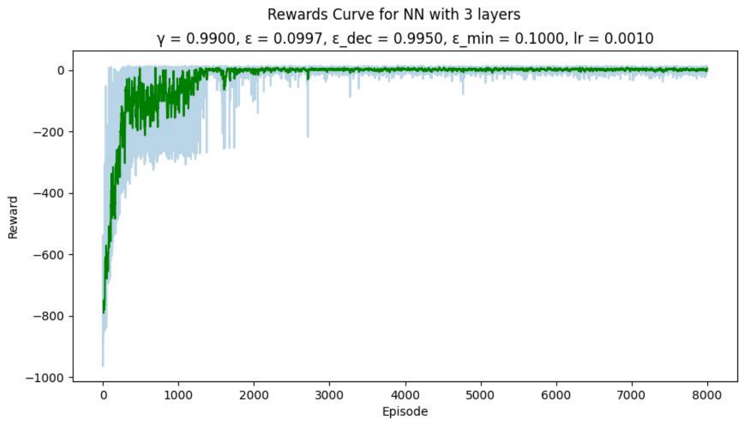
$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.995$

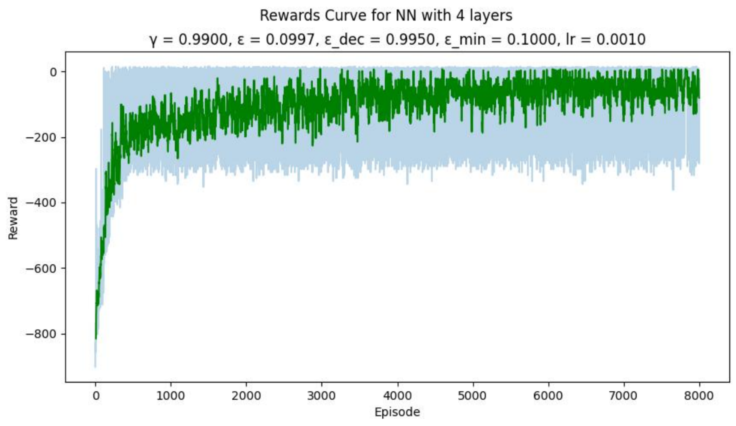


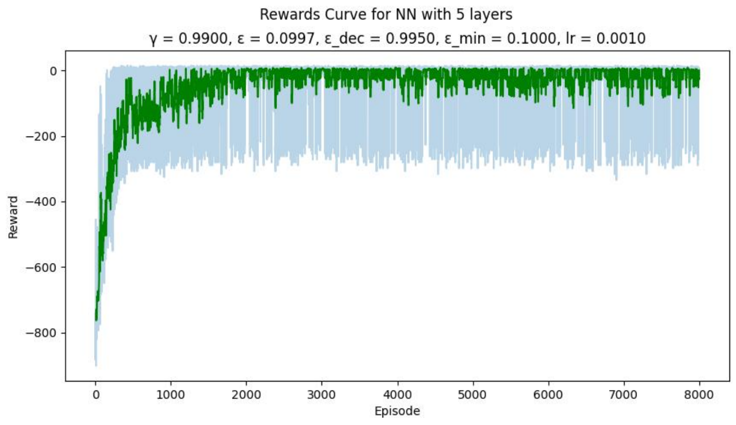
Observation on the test With 32 neurons per hidden layer, the agent with most relevant results is the one with a network with 5 layers, since it is sensible to variation of the usual parameters. A learning rate of 0.001 and ϵ decay factor of 0.995 the agent tends to 0 rewards so far with the increasing number of episodes and with a "noisy behavior". Adjusting a little ϵ_{dec}, it is possible to see that 0 value for rewards is reached earlier, but the "noise behavior" is still present. With a change of the learning rate to 0.002, the noise starts to fade.

4.3.4 Networks with 48 neurons for each hidden layer

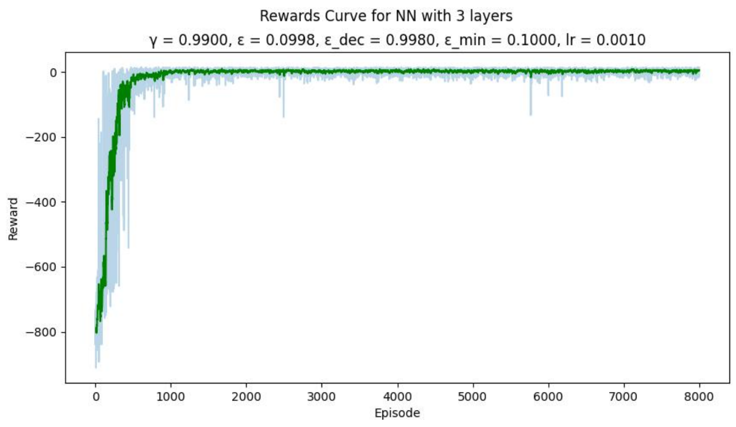
$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.995$

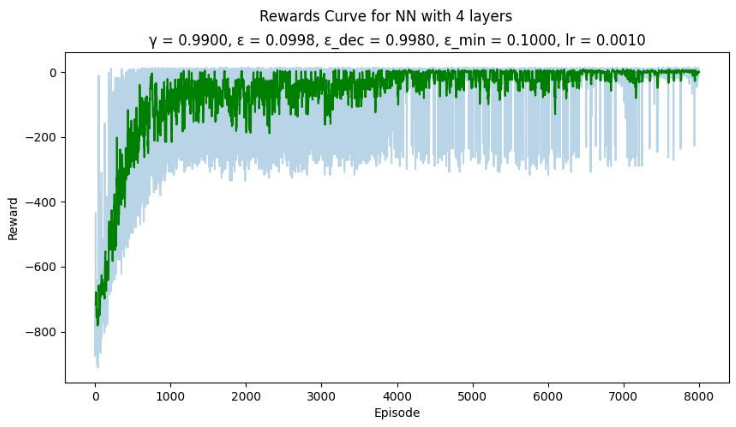


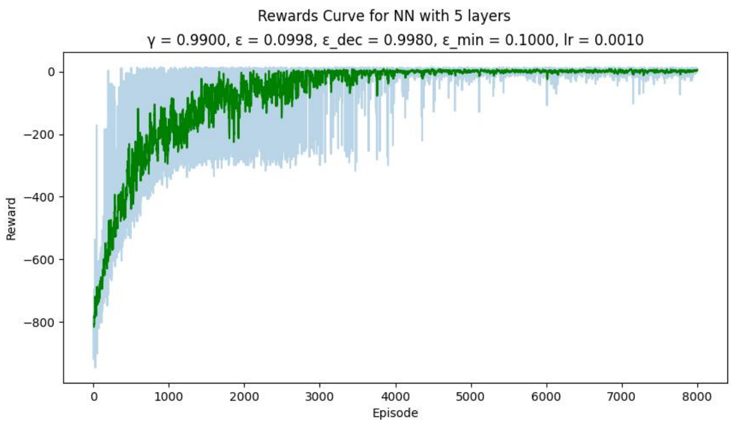




$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.998$


Rewards Curve for NN with 3 layers
γ = 0.9900, ε = 0.0998, ε_dec = 0.9980, ε_min = 0.1000, lr = 0.0010
Reward
Episode
0
−200
−400
−600
−800
0
1000
2000
3000
4000
5000
6000
7000
8000

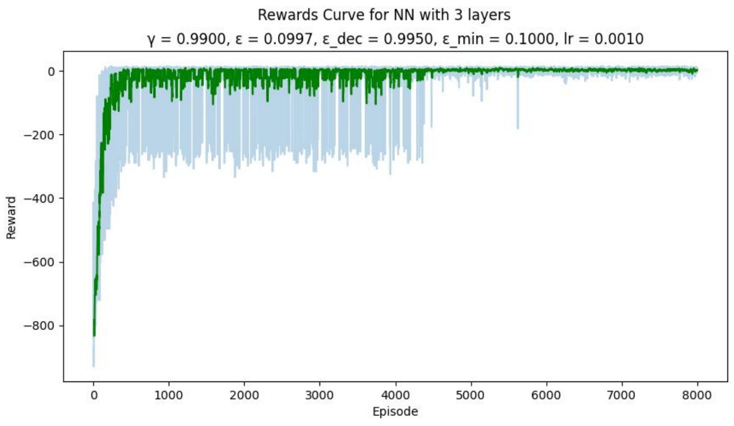

Rewards Curve for NN with 4 layers
γ = 0.9900, ε = 0.0998, ε_dec = 0.9980, ε_min = 0.1000, lr = 0.0010
Reward
Episode
0
−200
−400
−600
−800
0
1000
2000
3000
4000
5000
6000
7000
8000

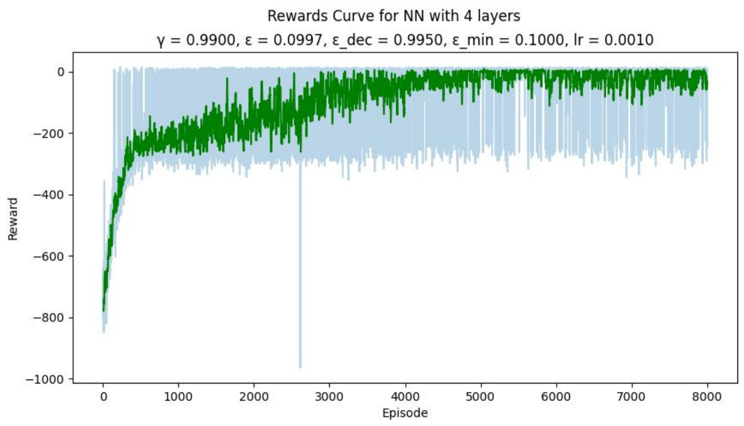


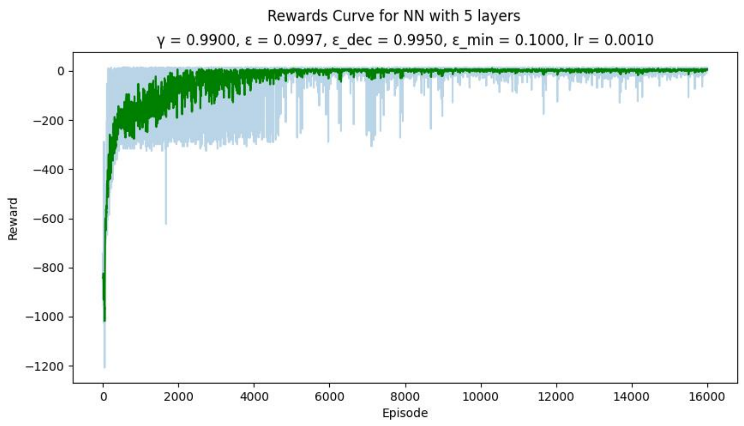
Observation on the test By the increase of the number of neurons for the hidden layers, the agents started to show better rewards in terms of rapidity and quality and also about adopting best policies for solving the environment.

4.3.5 Networks with 64 neurons for each hidden layer

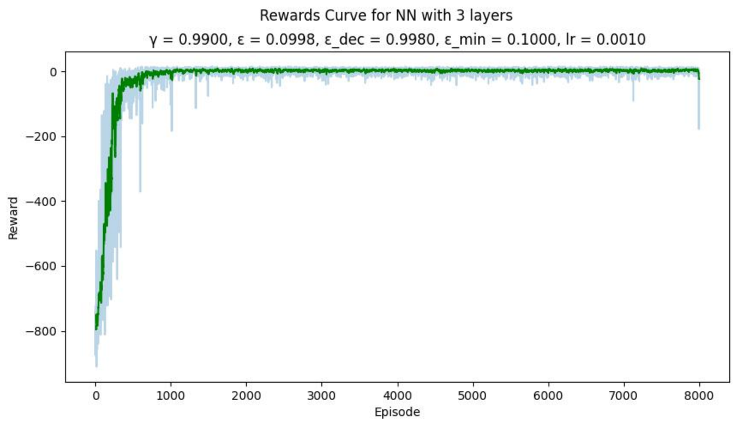
$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.995$

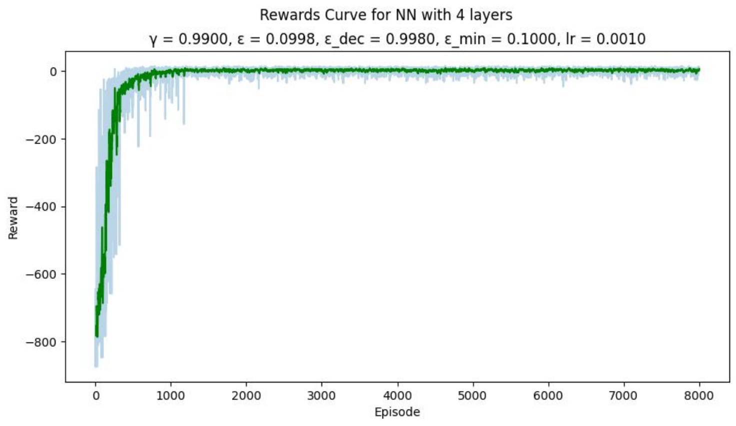


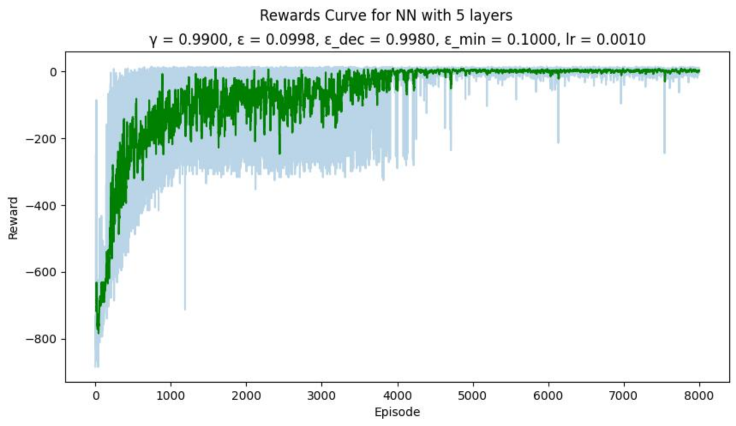




$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.998$


Rewards Curve for NN with 3 layers
γ = 0.9900, ε = 0.0998, ε_dec = 0.9980, ε_min = 0.1000, lr = 0.0010
Reward
Episode
0
−200
−400
−600
−800
0
1000
2000
3000
4000
5000
6000
7000
8000

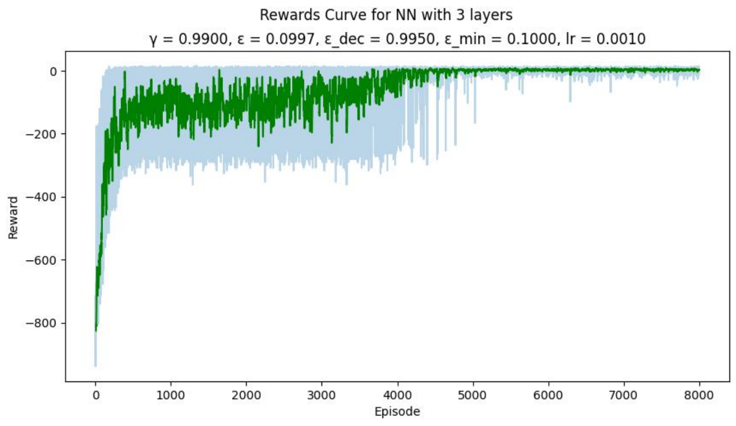

Rewards Curve for NN with 4 layers
γ = 0.9900, ε = 0.0998, ε_dec = 0.9980, ε_min = 0.1000, lr = 0.0010
Reward
Episode
0
−200
−400
−600
−800
0
1000
2000
3000
4000
5000
6000
7000
8000

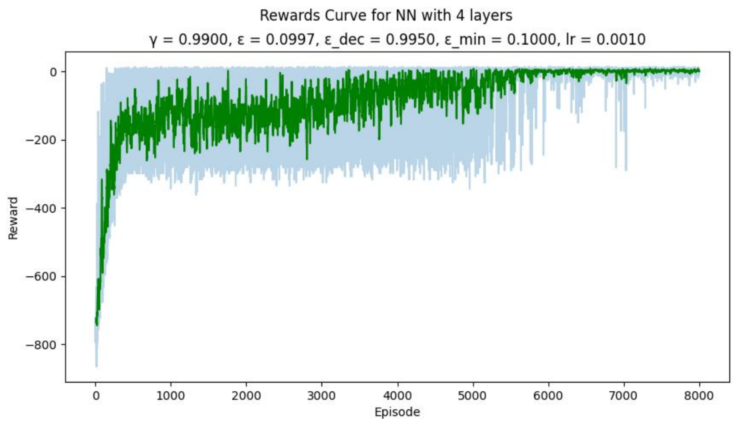


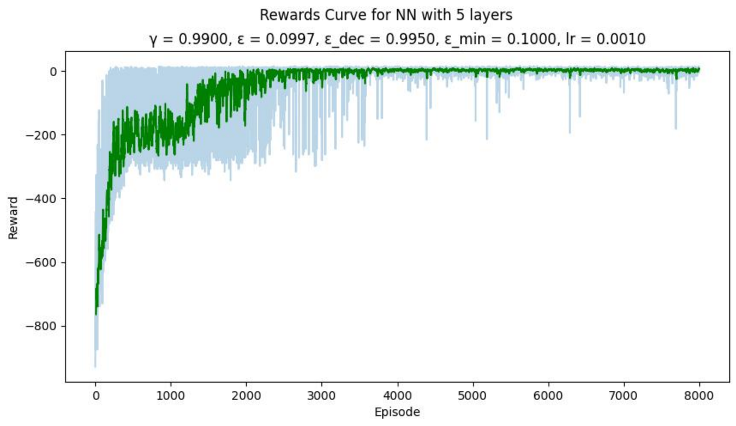
Observation on the test Accordingly with what seen in the previous test, adding more neurons to the layers generates a positive trend on the rewards curve on all the agents. A quite good behavior is registered on agents with networks having 3 and 4 layers with $\alpha = 0.001$ and $\epsilon_{dec} = 0.998$.

4.3.6 Networks with 96 neurons for each hidden layer

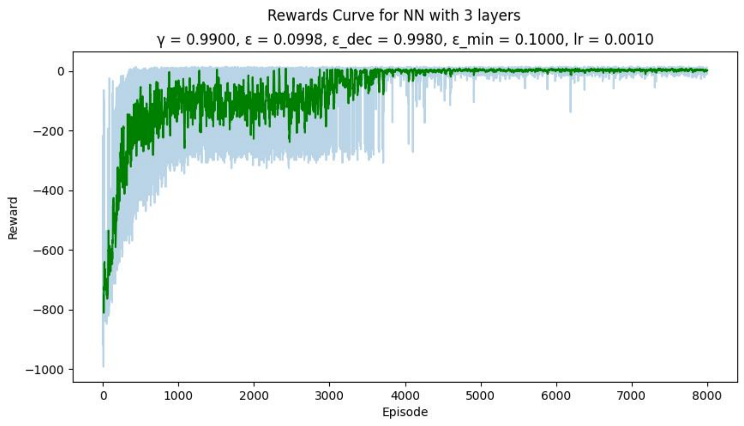
$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.995$

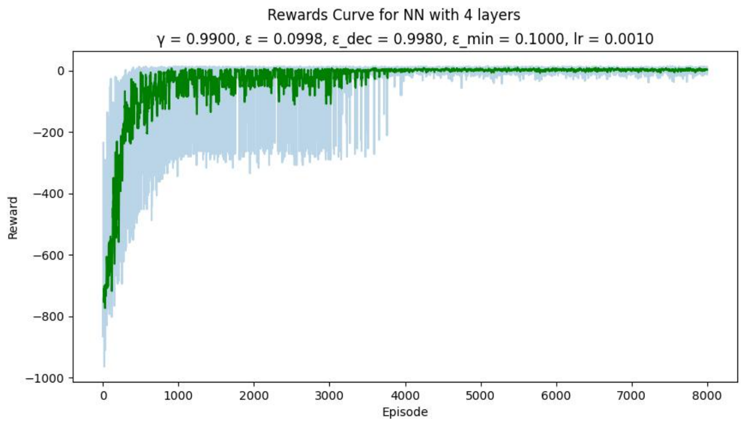


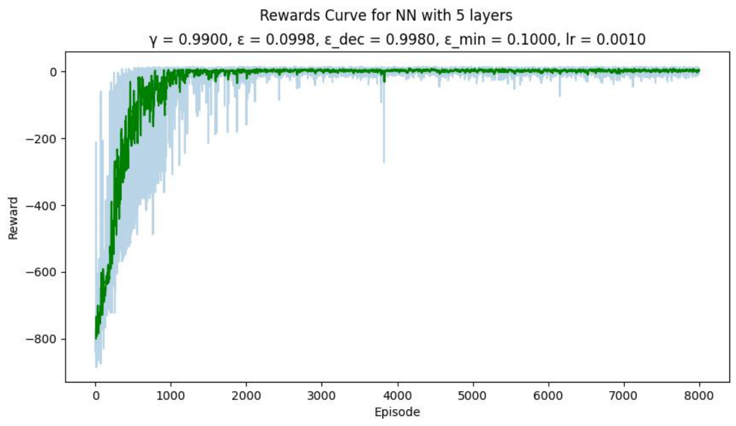




$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.998$


Rewards Curve for NN with 3 layers
γ = 0.9900, ε = 0.0998, ε_dec = 0.9980, ε_min = 0.1000, lr = 0.0010
Reward
Episode
0
−200
−400
−600
−800
−1000
0
1000
2000
3000
4000
5000
6000
7000
8000


Rewards Curve for NN with 4 layers
γ = 0.9900, ε = 0.0998, ε_dec = 0.9980, ε_min = 0.1000, lr = 0.0010
Reward
Episode
0
−200
−400
−600
−800
−1000
0
1000
2000
3000
4000
5000
6000
7000
8000



Observation on the test Increasing the number of neurons to 96 gives good results, but they aren't so good as for 64 neurons. The 0 value for the rewards is easily reached but it brings some noise on the curves.

4.4 General observations

On the basis of tests has been possible to show how the agents change their behavior with respect to the characteristics of the neural networks they use. Increasing the number of neurons for each hidden layer of the networks seems to give good benefits to the results and performances, even if going with an higher number of neurons add noise on the phase of the learning before the convergence to the discovery of an optimal policy. Overall, the best result are given by the agents using networks with 64 neurons per hidden layer (4.3.5).

5 Third solution: Neural Network with Target Network

5.1 Idea behind the solution

Starting from the previous idea of neural network, in this solution is adopted another network called "Target". The main goal of that is to make more stable and more precise the learning during the execution: instead of updating the network at each episode, it will be renewed the information inside the layers each a specific number of episodes. More in depth, the "Q-Network" will be used for generating Q-values by the current state and for executing actions on the environment, while the "Target Network" is used for retrieving the next Q-values without being used for other computations.

5.2 Implementation

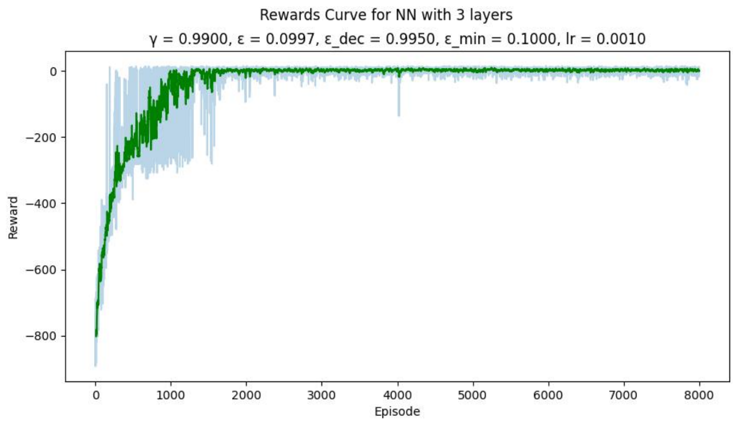
With respect to the previous implementation, there are only few differences. The first one relies on the class `Agent`, in particular inside the function `init` and function `replay`: instead of using only one network, there are the previously presented `q_network` and `target_target` that are used for retrieving Q-values on current state and next Q-values related to possible new states respectively. The Q-Network is updated with the same frequency of the previous case, meanwhile the Target network is updated each 10 episodes on the basis of the values stored inside Q-Network.

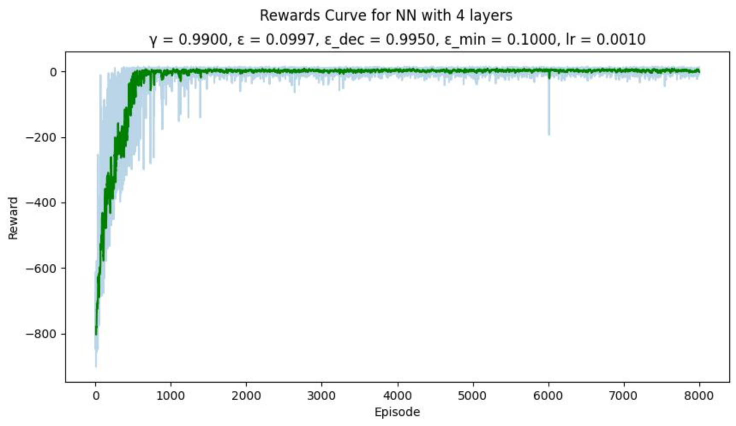
5.3 Experiments

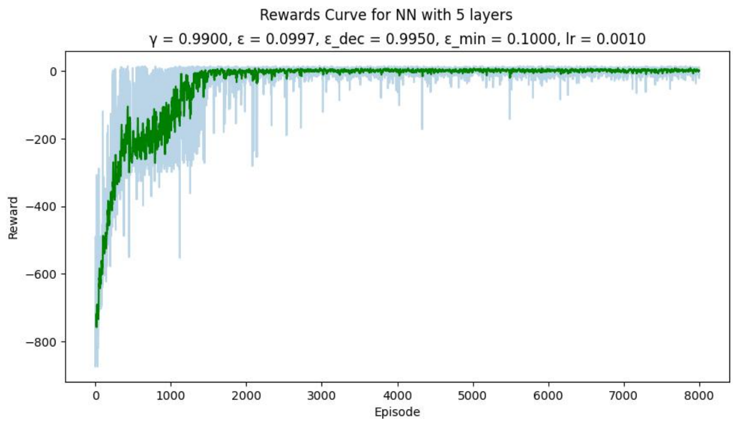
The tests done for this solution, in general, returned better results with respect to the implementation without the target network. The changes on parameters were about the width of hidden layers and ϵ_{dec}. Only plots with "significant" differences are reported since, in many situations, the performances obtained are quite similar.

5.3.1 Networks with 16 neurons for each hidden layer

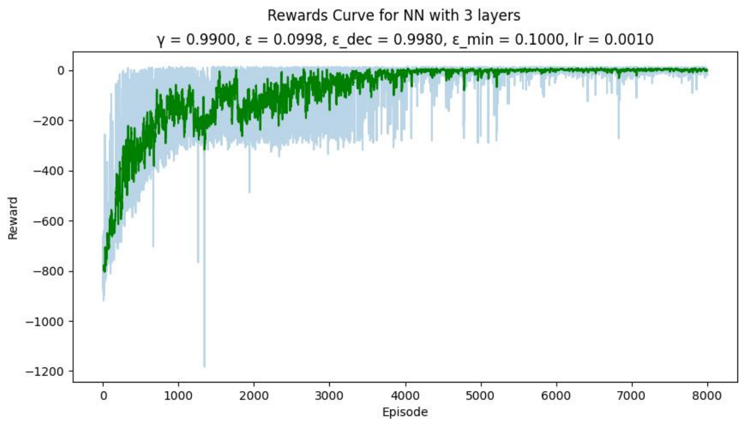
$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.995$

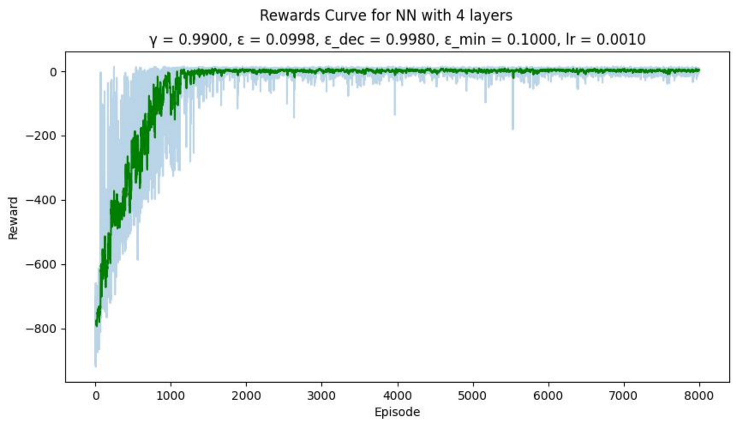


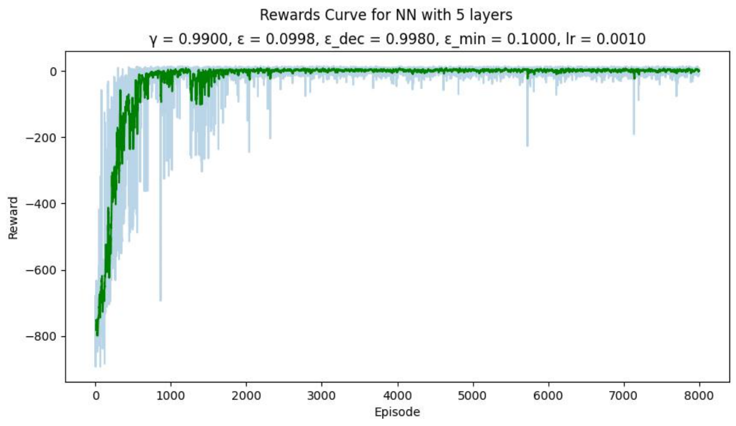




$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.998$



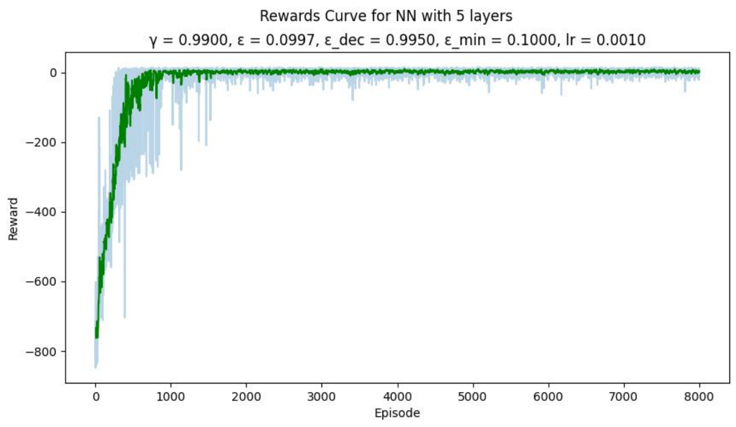




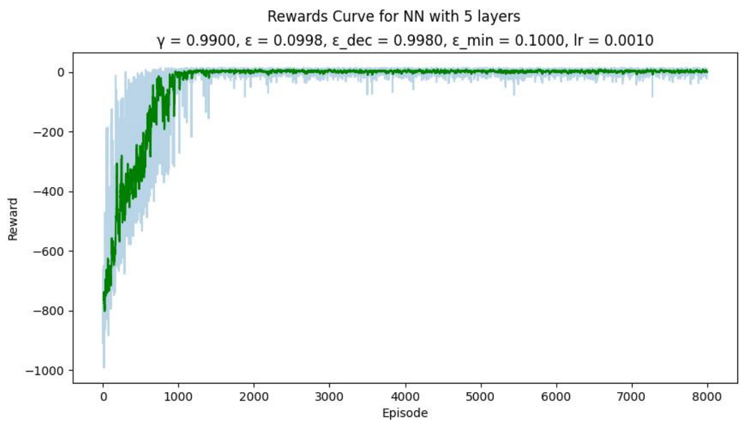
Observation on the test Setting to 0.998 the decay factor, as suggested by the theory, the agents seems to have a little delay to find a good policy for solving the environment if compared with the tests with $\epsilon_{dec} = 0.995$ where seems to learn a little bit faster.

5.3.2 Networks with 24 neurons for each hidden layer

$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.995$



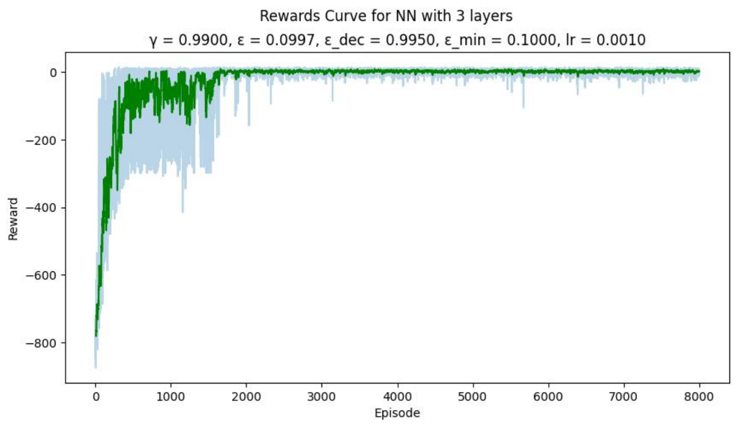
$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.998$



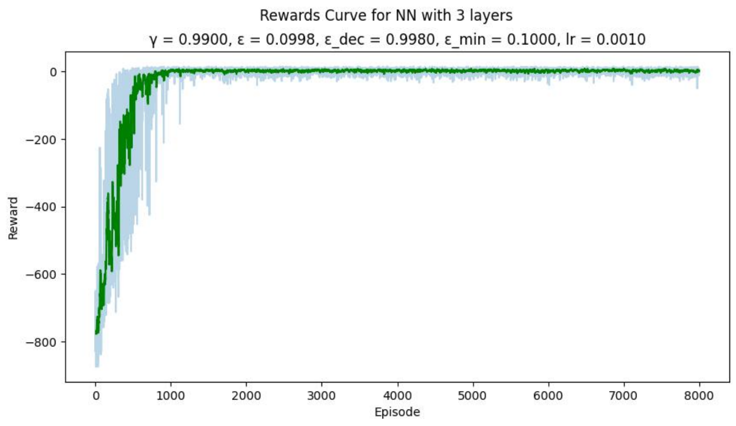
Observation on the test There is visible difference on the tests only on the agents using the network with 5 layers. In fact, with $\epsilon_{dec} = 0.998$ there is a slightly slower curve on the rewards on finding good values, since the agent can explore more.

5.3.3 Networks with 32 neurons for each hidden layer

$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.995$



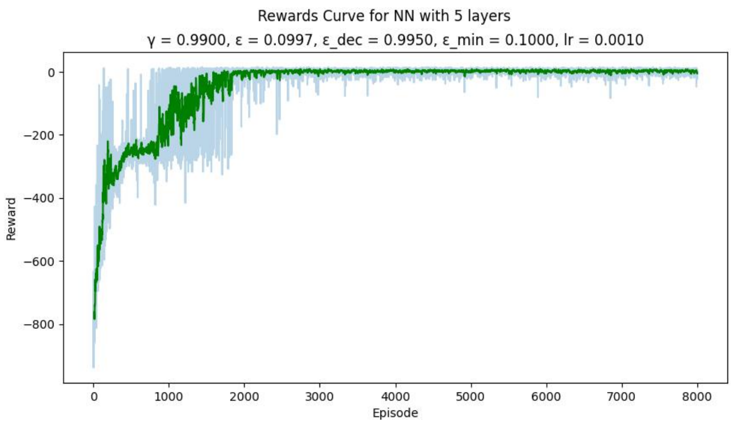
$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.998$



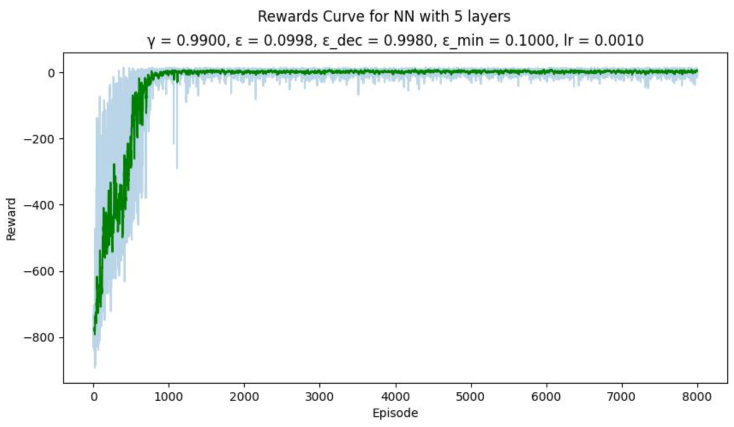
Observation on the test In contrast with what has been seen in the previous two tests, in this situation is possible to observe that with a higher decay factor for ϵ there is a trend to reaching good rewards with approximately 1000 episodes more than the case with lower decay.

5.3.4 Networks with 48 neurons for each hidden layer

$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.995$



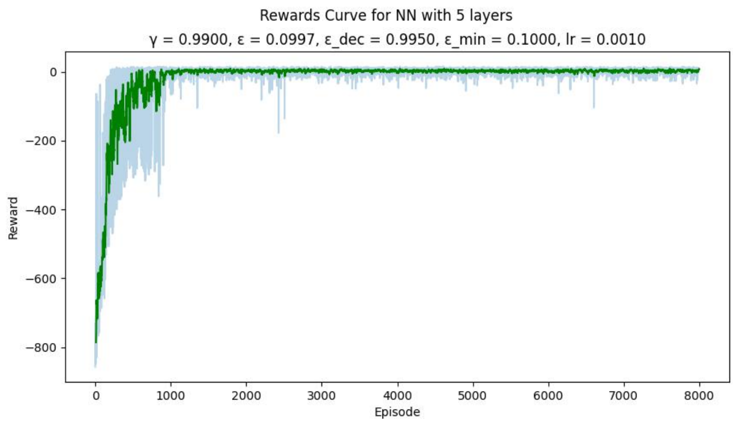
$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.998$



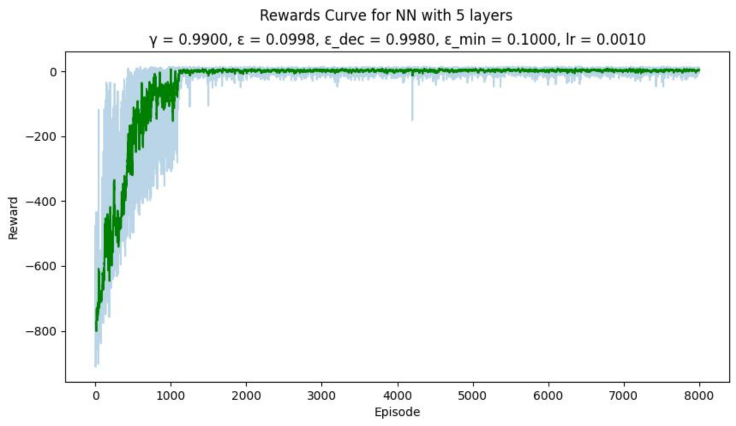
Observation on the test Also in these tests, better performances are achieved with $\epsilon_{dec} = 0.998$ and with a longer exploration phase along the learning and confirming the positive trend.

5.3.5 Networks with 64 neurons for each hidden layer

$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.995$



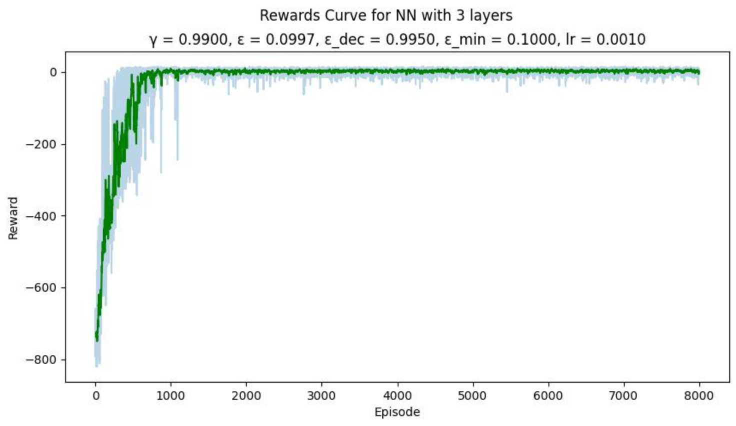
$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.998$

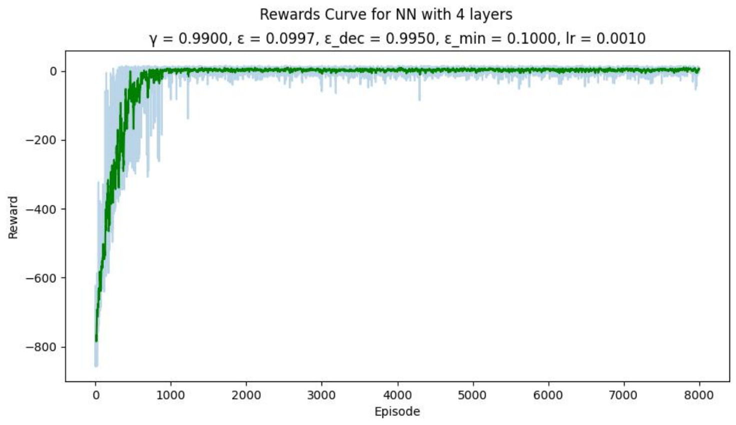


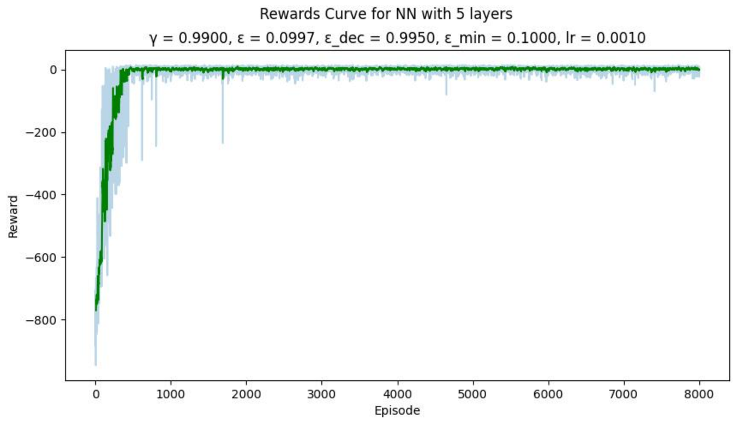
Observation on the test About the test with $\epsilon_{dec} = 0.998$, it is possible to evince two distinct phases before reaching an optimal policy: in the first one there are lower total rewards while in the second one there is a trend to better results, even if with some "noisy spikes". A slightly different situation there is on the other plot where the reward curve follows a "smoother" behavior with an optimality in lower number of episodes.

5.3.6 Networks with 96 neurons for each hidden layer

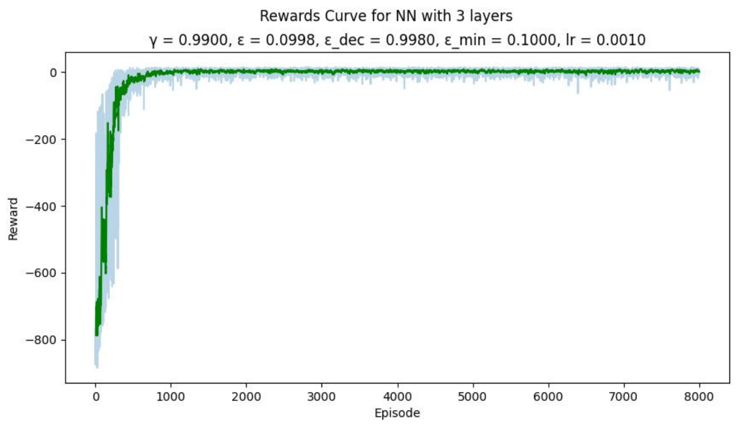
$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.995$

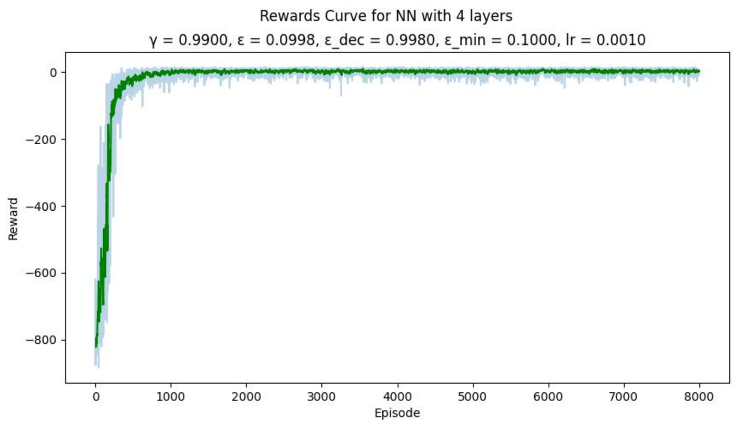


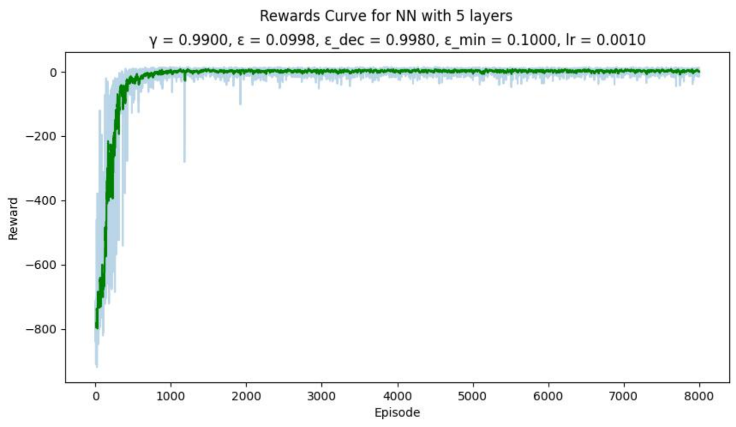




$\alpha = 0.001$, $\gamma = 0.99$, $\epsilon = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{dec} = 0.998$


Rewards Curve for NN with 3 layers
γ = 0.9900, ε = 0.0998, ε_dec = 0.9980, ε_min = 0.1000, lr = 0.0010
0
−200
−400
−600
−800
Reward
0
1000
2000
3000
4000
5000
6000
7000
8000
Episode


Rewards Curve for NN with 4 layers
γ = 0.9900, ε = 0.0998, ε_dec = 0.9980, ε_min = 0.1000, lr = 0.0010
0
−200
−400
−600
−800
Reward
0
1000
2000
3000
4000
5000
6000
7000
8000
Episode


Rewards Curve for NN with 5 layers
γ = 0.9900, ε = 0.0998, ε_dec = 0.9980, ε_min = 0.1000, lr = 0.0010
0
−200
−400
−600
−800
Reward
0
1000
2000
3000
4000
5000
6000
7000
8000
Episode

Observation on the test The networks with 96 neurons per hidden layer seems to allows the agent to find efficiently and rapidly the best policy for solving the environment also with a great stability on the rewards curve. There are no significant differences between all the variation of number of layers and hyperparameters.

5.4 General observation

Thanks to the adoption of the "Target" network is possible to get good results on the rewards also with few neurons per layer. In contrast with the previous solution, there are no strong differences between "wider" and "thinner" networks, even if the widest ones seems to suffer less the "noise" on the rewards curve.

6 Conclusions

In the report are reported different methods and approaches from Machine Learning for solving a discrete environment. In general, the adoption of the reinforcement learning based on "ϵ-greedy" leads to some tuning of the parameters used by the agent for reaching the goal of finding the best policy for correctly terminating the episodes.

The first solution, so called "Tabular Q-Learning", requires a lot less resources in terms of computational power and time. Even if it is "easier" to implement, without the need of too complex functions and libraries, in this environment allows to retrieve very good results in terms of rewards with most of the different parameters tested.

Instead of directly storing the information inside the table and manually retrieving them, the agents with neural networks work differently. Once an action is performed, the data are sent to the network and all the layers of it are in charge to manipulate them for a better analysis. Each layer, working with its own neurons, keeps track of all the weights the data allow to generate and passes them, when possible, to the other layers. By changing the depth and the width of the network with the relative parameters of the agent, it was possible to achieve different results.

The adoption of a "single" network inside an agent leads to a continuous update on the weights of each layer From the tests, it emerges that this type of solution has more difficulty to find the optimal policy with a lower number of neurons per hidden layer. Some improvements of the results came from the change of the decay factor of ϵ and the learning rate α, but the best ones are available with wider networks as the 64 and 96 neurons "versions".

Pairing the previous network with another network, the "Target" one, gives benefits to all the agents that uses it. As described in its implementation, this solution allows to have a more stable learning with less noise and higher speed on the discovery of the optimal policy. This can be seen just looking at the rewards curve plots. This is possible by a lower frequency on the updates on the network; in fact, the "future" Q-values are taken directly from the target network, the one that receives updates of the state each a fixed number of episodes (in this implementation was set to 10). Good results are available starting from thinner networks: comparing the rewards obtained by the agent with "16 neurons" networks in both the solutions leads to a higher performances for the second version. Same happened for the other tests with a peak in results with the agent using the network with 96 neurons/hidden layer.