Assignment 1 – From Q-learning to Deep Q-learning (DQN)

Submitted by :

Gur TBD

Gal Benmaman 205633589

## Section 1 – Tabular Q learning

1. Value iteration is an algorithm that calculate the optimate state values V(s) by iteratively updating them using bellman optimality equation:

It means we need to have information about the probability of each state and reward based on the back state and action. If we don’t have the transition probability, the algorithm will not work.

1. Model-free methods like SARSA and Q-Learning don’t require the transition probability- the probability of getting from the models interact with the environment and observe the state and actions. Then update the Q value of the state and action according to the current policy/exploration which require the next state and the chosen action.
2. Both algorithms, SARSA an Q-Learning, are model-free methods, SARSA is an On-policy model, it updates the Q-Value based on the actions the agent actually takes according to the policy. While Q-Learning is an off-policy model, it updates the Q-value by the optimal action which received the highest reward. Q- learning will choose the best reward action, even if the probability of the action is very low (like winning a lottery prize- high reward with low probability)
3. The balance of exploration and exploitation by epsilon-greedy allows the agent to converge to the optimal action-state, (exploitation), but in other hand it allow the agent to explore new actions, which vary the current action to explore even better actions with highest reward (explore). if we use greedy search the agent is in risk to converge in local-optima action

### Report – Q learning

We implemented Q learning in Frozen lake by Gymnasium library. We used the FrozenLake-v1 with the next parameter: is\_slippery = True. The implantation attach in file section1.py.

For hyper parameters tuning we used the following grid:

*HYPERPARAMS\_grid = {*

*'alpha' : [0.3,0.2,0.1,0.05,0.01,0.005],            # Learning rate*

*'gamma' : [0.9999,0.99,0.97,0.95,0.9],          # Discount factor*

*'epsilon' : [1.0],         # Initial epsilon for exploration*

*'epsilon\_decay' :[0.995,0.99,0.9], # Decay rate for epsilon*

*'epsilon\_min' : [0.01],     # Minimum epsilon*

*'n\_episodes' : [5000],    # Total episodes*

*'max\_steps' : [100]        # Max steps per episode*

*}*

We chosed high epsilon decay rates to allow the model gradually convergence to the minimum epsilon value This slow reduction gives the model sufficient steps to explore different actions, enabling it to improve its policy effectively .

To evaluate the performance of the agent we used 2 Main KPIs :

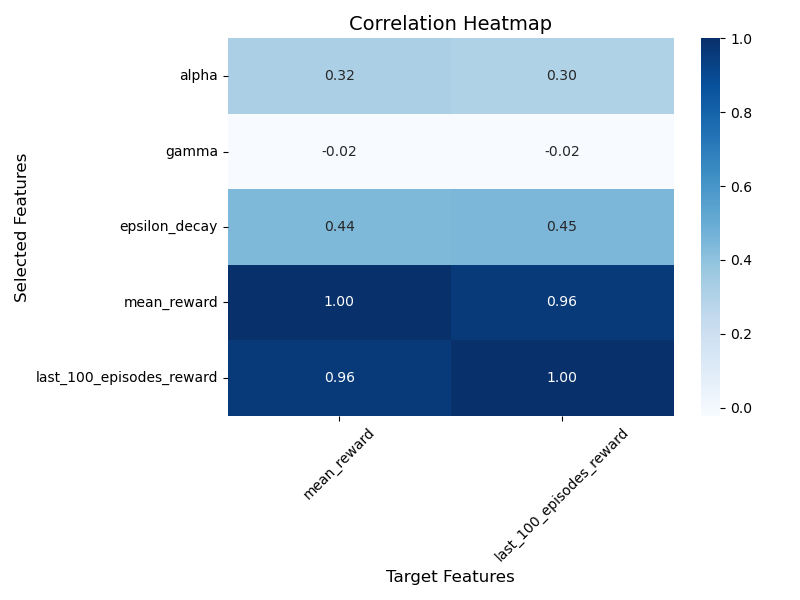
1. Mean reward : The average reward across all episodes. This metric provides insight into how quickly the model converges to an optimal policy and how well it balances the speed of convergence with the quality of the learned policy.

A screenshot of a black screen

Description automatically generated

1. The average reward obtained in the last 100 episodes. This metric indicates the stability and performance of the agent after training is complete





Both KPIs are highly correlated, which suggests that models with higher mean rewards also tend to have better stability in the last 100 episodes. Moreover highe learning rate (alpha) and slower epsilon decay rates tend to help the agent, small Epsilon decay rates lead to quickly reduction of the epsilon ( in this case the agent lack of exploration during the learning process.

#### Best Model evaluation

Given the short training time (a few seconds per run), we selected **Last 100 Episodes Reward** as the primary KPI for identifying the best model. The best hyperparameters were:

*Best HYPERPARAMS = {*

*'alpha' : [0.3],            # Learning rate*

*'gamma' : [0.97],          # Discount factor*

*'epsilon' : [1.0],         # Initial epsilon for exploration*

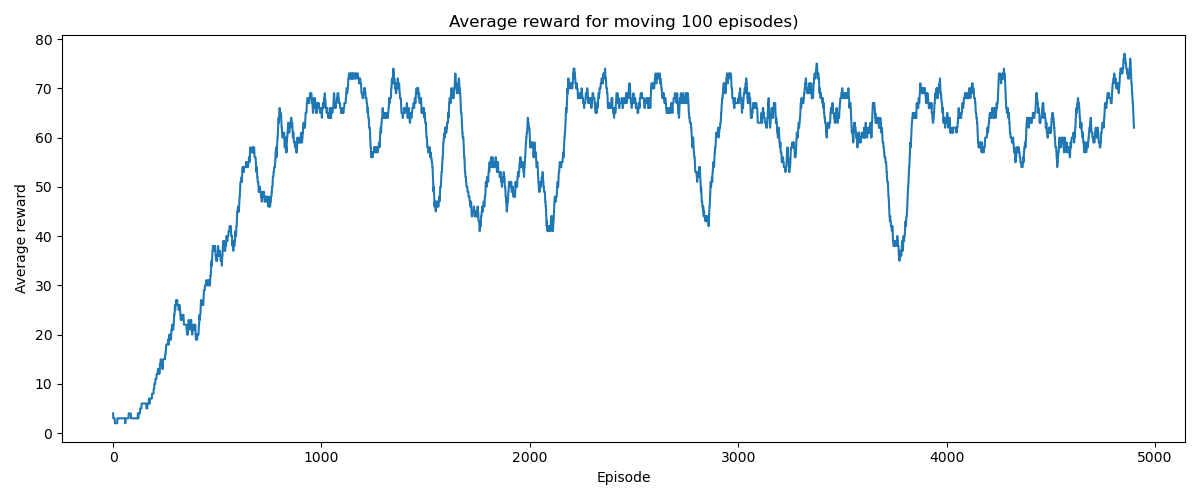
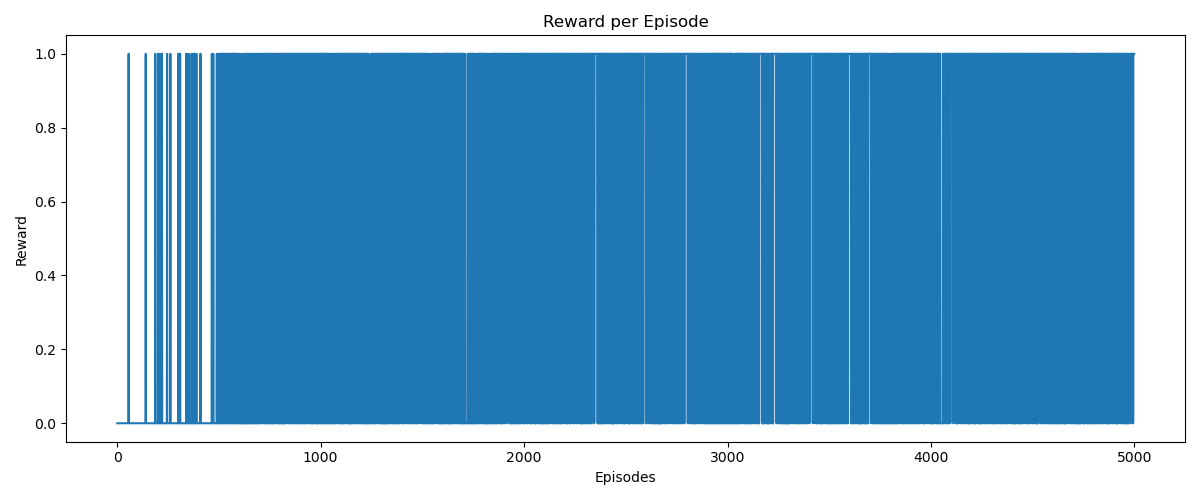
*'epsilon\_decay' :[* *0.995], # Decay rate for epsilon*

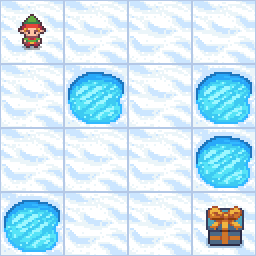
*'epsilon\_min' : [0.01],     # Minimum epsilon*

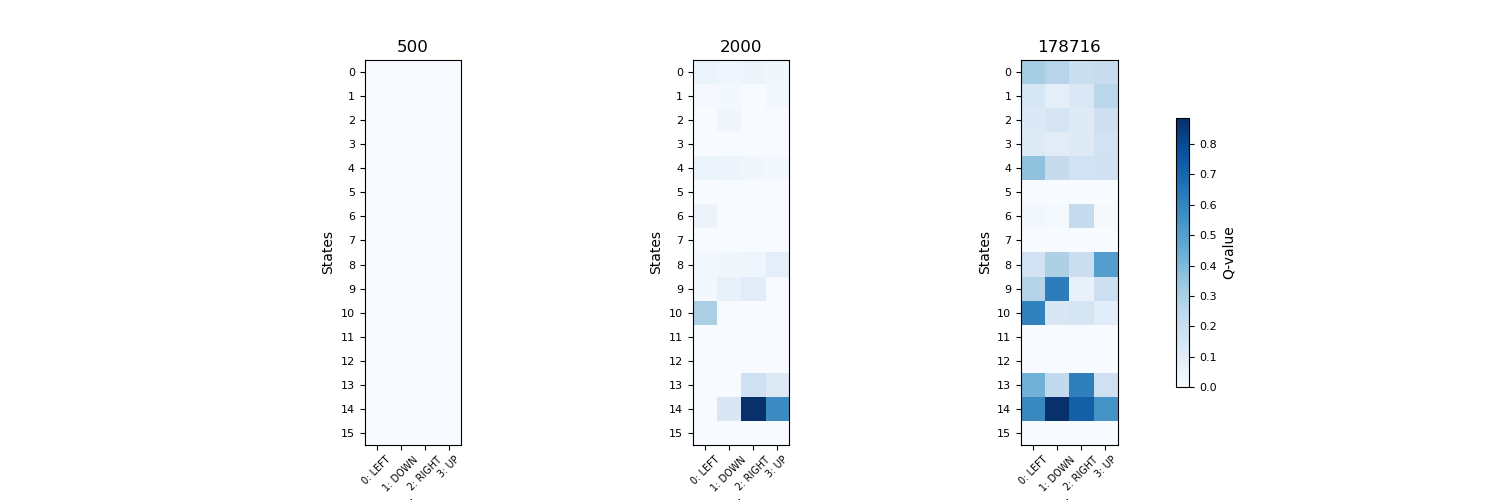
*'n\_episodes' : [5000],    # Total episodes*

*'max\_steps' : [100]        # Max steps per episode*

*}*

**

**



We can see that the best model convergence after around 700-800 episodes to a success rate of 0.8, consider the minimal epsilon of 0.01 and slippery. The graph Reward per spisode shows that after 500-600 episodes the model consistently reach the goal. the Q table plot show the Q table in 3 time points: 1) 500 steps 2) 2000 steps 3) final step. We can see the after 500 steps the agent still has minimal knowledge of the environment, after 2000 steps The agent demonstrates improved understanding, especially in states near the goal state , and the final q table show much more clear and comprehensive policy according to each state in the game.

## Section 2 – Deep Q-learning

1. Sampling experiences in random order from the replay memory improves the generalization of learning. By using non-consecutive states , the agent learn to deal with varies states, otherwise the agent will be at risk to overfit the action of each sets of consecutive set of states
2. By using two networks in Deep Q-Learning—a Q-network, which is updated at every step, and a target network, which is updated gradually (for example in each c steps or using TAU)—we stabilize the learning process. The gradual updates to the target network smooth the Q-value predictions and prevent the loss function from excessively influencing weight optimization. Additionally, when the same network is used to both predict and compute target Q-values, the targets change as the network updates, leading to instability. By decoupling the target network (responsible for computing the target Q-values) from the prediction network (which learns to approximate the Q-values), we mitigate this "moving target" problem and create a more stable training process.

### Report – Deep Q learning

We implemented Deep Q-Learning (DQN) using two neural networks with different architectures (3 layers and 5 layers). A hyperparameter tuning pipeline was employed to optimize the agent's performance. The implementation, including the hyperparameter search pipeline, is located in the file DQNcartpole.py.

#### Network structure

We implemented two networks:

1. 3 layers network:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Layer** | **Input dimension** | **Output dimension** | **Activation function** | **Dropout** |
| 1 | Input dimension | 128 | Gelu | 0 |
| 2 | 128 | 64 | Gelu | 0.1 |
| 3 | 64 | 2 |  |  |

1. 5 layers network:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Layer** | **Input dimension** | **Output dimension** | **Activation function** | **Dropout** |
| 1 | Input dimension | 128 | Gelu | 0 |
| 2 | 128 | 128 | Gelu | 0 |
| 3 | 128 | 128 | Gelu | 0 |
| 4 | 128 | 64 | Gelu | 0.1 |
| 5 | 64 | 2 |  |  |

Optimizer: ADAMW

Loss function: Mean squared error (MSE)

#### Hyper parmeter tuning

We performed grid search optimization for the following parameters:

hyperparams = {

    "LR": [0.01,0.001,0.0001],

    "GAMMA": [0.95,0.97, 0.99],

    "batch\_size": 32,

    "tau": [0.01,0.1,0.5],

    "replay\_memory\_size": 16000,

‘EPS\_DECAY’ = 2000,

    ‘EPS\_START’ = 0.9,

    ‘EPS\_END’ = 0.001

}

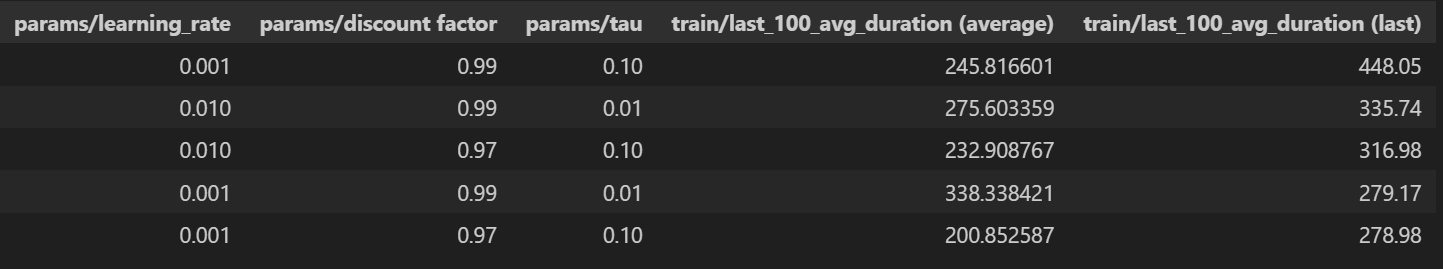
We focused on optimizing agent-specific parameters, as we believe they have the most significant impact on solving the problem and provide insights into their influence.

We selected the last average reward over the final 100 episodes as our metric. This reflects both the performance and stability of the final agent.

We used the Neptune.ai framework to store and analyze results. Given more time and computational resources, we would further optimize network parameters such as the number of weights, dropout rate, and batch size

#### Hyper parameter tuning analysis

The best model of 3 layers network:

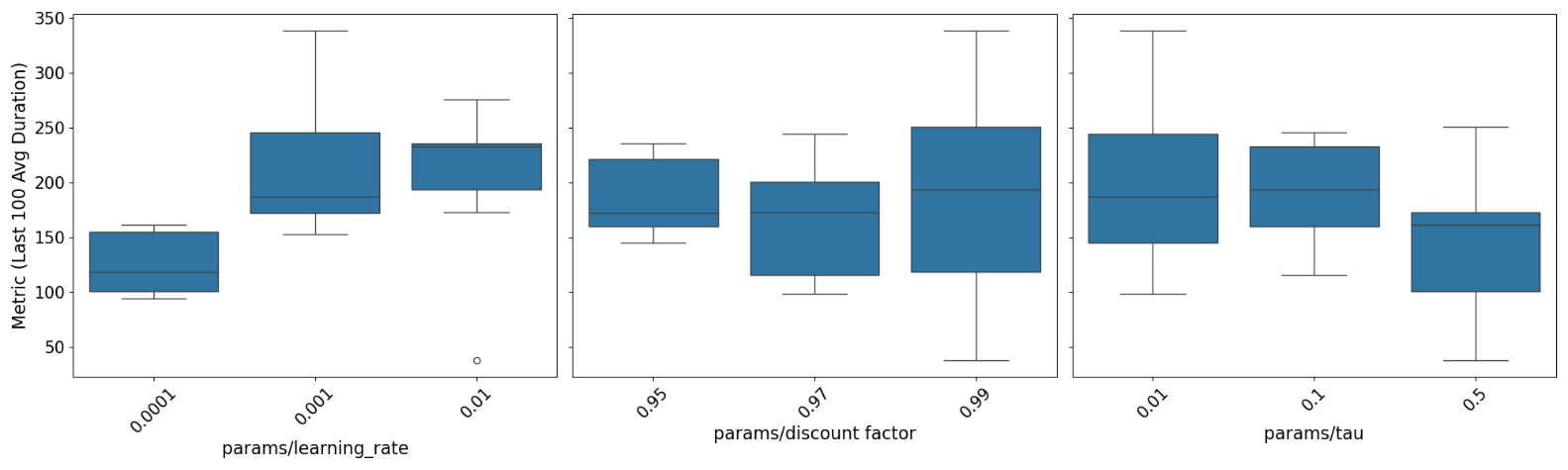


The best model has the next parmeters:

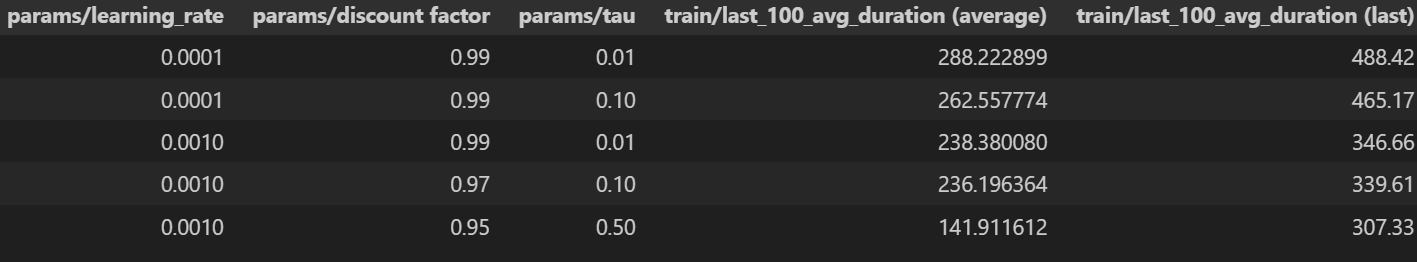
hyperparams = { "LR": 0.001, "GAMMA": 0.99, "batch\_size": 32, "tau": 0.1, "replay\_memory\_size": 16000, "EPS\_DECAY": 2000, "EPS\_START": 0.9, "EPS\_END": 0.001 }

The best model of 5 layers network:

**Hyper-parameter comparison : 3 layers model**



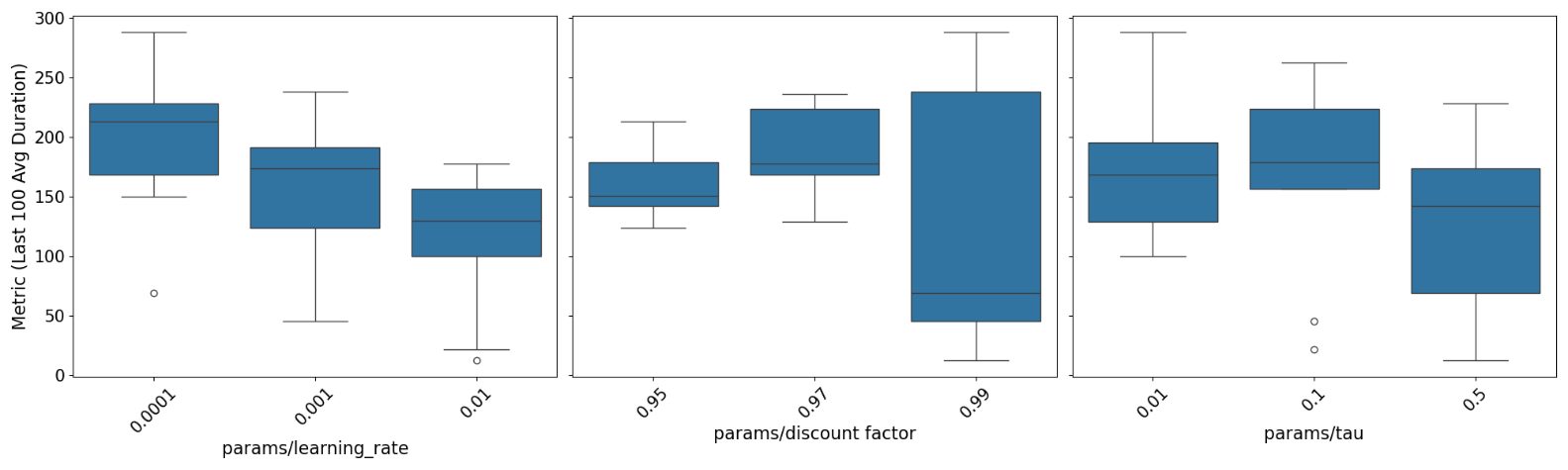
The best model of 5 layers network



The best model has the next parmeters:

hyperparams = { "LR": 0.0001, "GAMMA": 0.99, "batch\_size": 32, "tau": 0.01, "replay\_memory\_size": 16000, "EPS\_DECAY": 2000, "EPS\_START": 0.9, "EPS\_END": 0.001 }

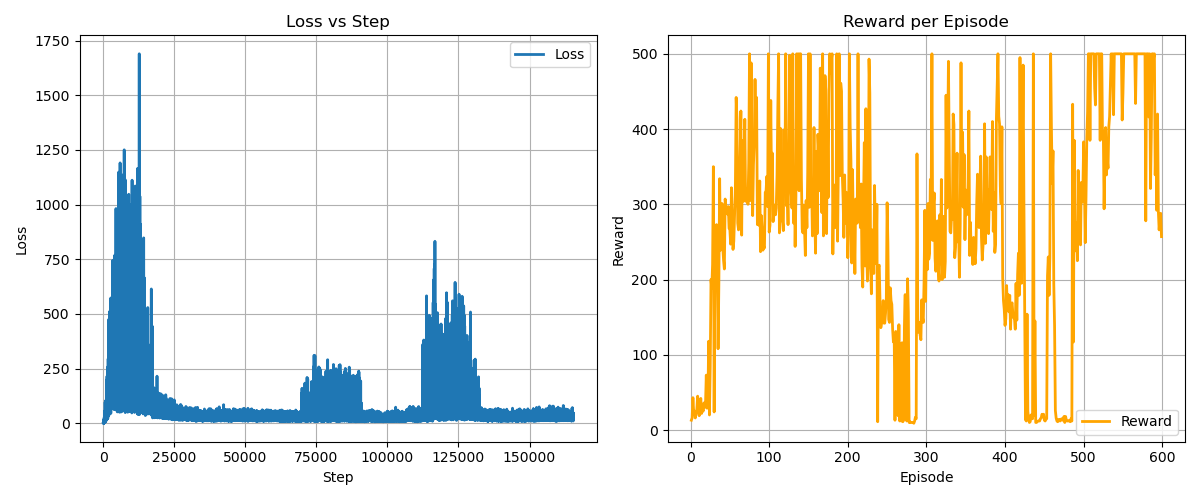
**Hyper-parameter comparison : 5 layers model**



We conducted 54 experiments (27 agent parameter combinations × 2 network sizes) and observed that the 5-layer network achieved better performance, with a mean reward score of 488 compared to 448 for the 3-layer network. This suggests that the 5-layer network is able to extract more information from the state space and produce better action decisions. Additionally, we found that a high update rate negatively impacted performance for both networks. The discount factor and learning rate showed different behaviors across the networks: in the 5-layer network, the best-performing discount factor was 0.97, while no single discount factor consistently outperformed the others in the 3-layer network. Interestingly, we observed high variability in rewards when using a discount factor of 0.99 in both models. Regarding the learning rate, the 3-layer network performed better with a higher learning rate, while the 5-layer network benefited from a lower learning rate, which improved its overall performance.

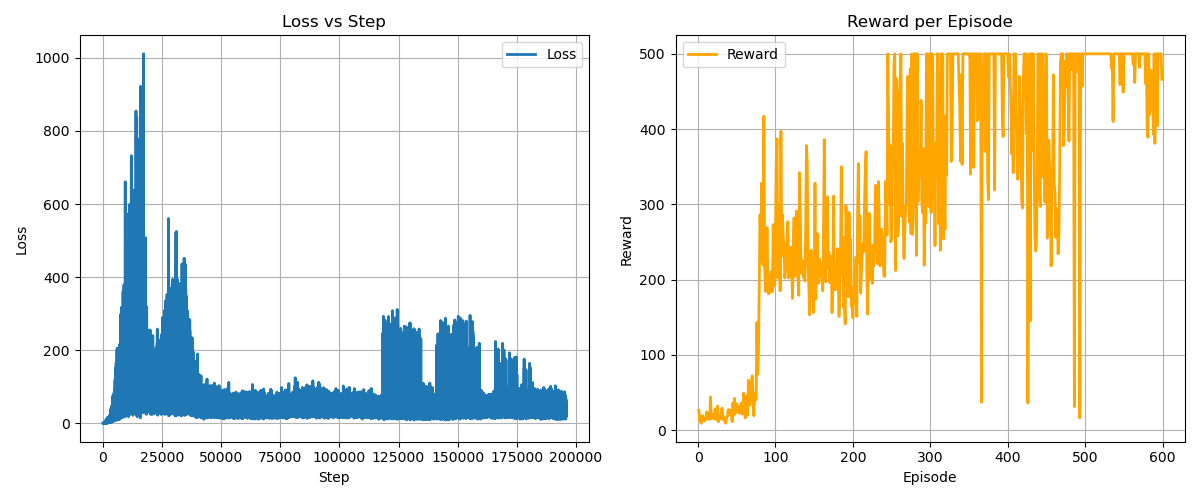
#### Model evaluation

3 layers model:



First episode to each mean average 100 episode of 450: the highest mean reward over the last 100 episode is 449, in episode 596

5 layers DQN:



First episode to each mean average 100 episode of 450: 401

All the experiments metadata are stored in output/section2 : there are 2 folders-: for both of the networks, each folder contains the hyper parameters optimization, moreover the evolution of the best model: loss per step, reward per episode and moving 100 average per episode.

## Section 3 – Improved DQN

Tbd – report of the script