****

**Ben-Gurion University of the Negev**

**Faculty of Engineering Sciences**

**Department of Software and Information systems**

**Deep Reinforcement Learning**

**Assignment #1**

**By**

**Oz Elbaz – 204388763**

**Ido Gurevich - 205478068**

**Section 1 - Tabular Q-learning:**

**Theoretical questions**

1. Value iteration is model-based method, meaning it requires **full knowledge** of the environment’s dynamics to determine the optimal solution. Moreover, the algorithm assumes a **finite** set of states and actions. Both are not the case for an environment with unknown or complicated dynamics. Hence, it cannot be implamented for this type of environments.
2. Model-free methods, don’t require knowledge of the **environment's dynamics**. Instead, they learn **directly** from interacting with the environment. These methods rely on trial and error to estimate the value of states or state-action pairs.
3. **SARSA** is an on-policy method that updates the Q-value based on the actual action taken in the next state following the current policy, reflecting the agent's real behavior. In contrast, **Q-learning** is an off-policy method that updates the Q-value using the maximum possible future reward, assuming optimal actions. This difference means Q-learning often converges faster but may disregard exploratory actions, unlike SARSA.
4. Acting greedily prioritizes short-term rewards but risks the agent getting stuck in suboptimal routines. A decaying epsilon greedy strategy balances exploration and exploitation by encouraging more exploration early in learning and gradually shifting towards greedy actions as the agent gains confidence in its knowledge.

**Q-learning agent:**

In this section, we implemented a Q-learning agent algorithm. The agent interacted with the environment FrozenLake-v1. First, we created a lookup table of the q-value for each state-action pair(16\*4=64) and initialize those values to zero.

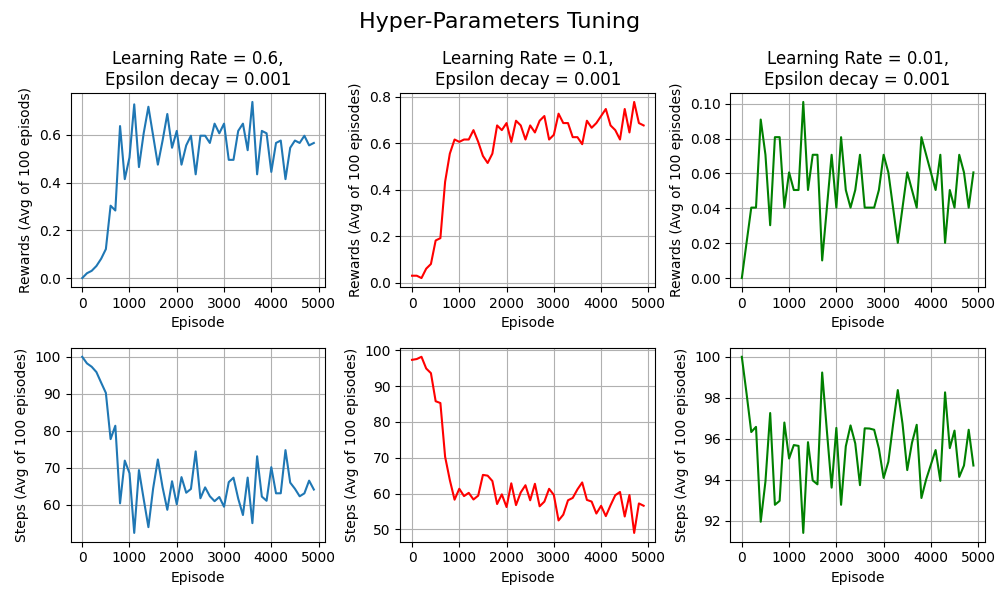


Figure 1: Hyperparameters tuning for the QL agent

Second, to choose the best hyperparameters we ran a few different combinations of hyperparameters. As can be seen in the Fig.1 an example for one attempt of hyperparameter tuning over the learning rate. As you can see the agent who outperform the rest was the agent in the middle panel with those hyperparameters:

* Learning rate: 0.1
* Epsilon decay: 0.001
* Discount factor: 0.99
* Initial epsilon: 0.8
* Final epsilon: 0.01

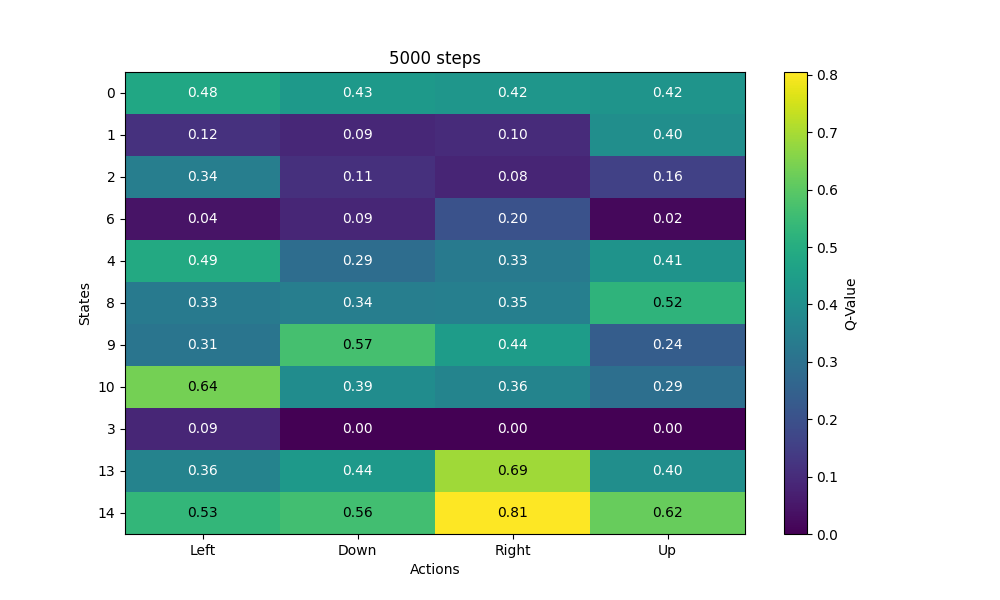
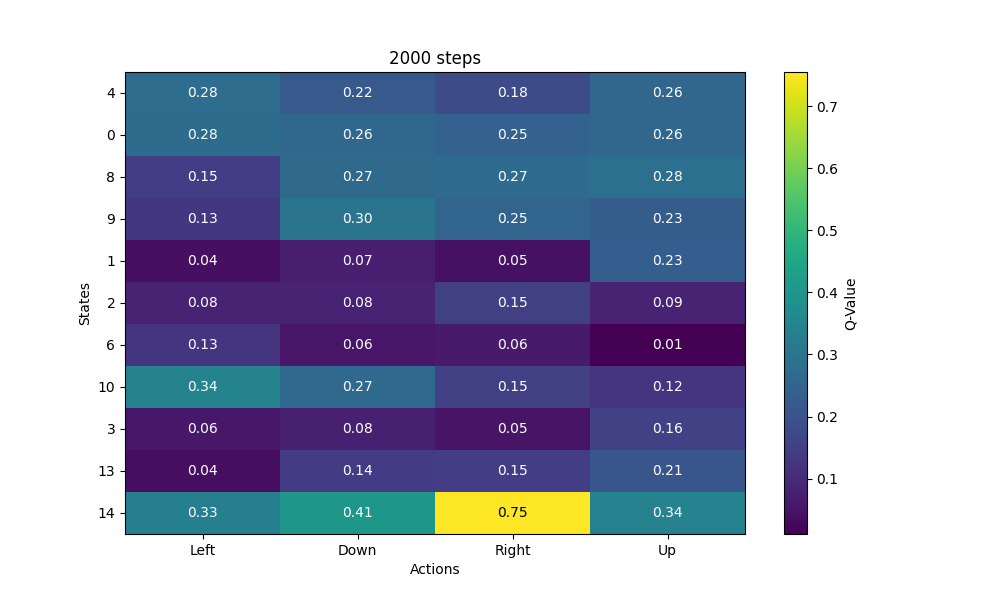
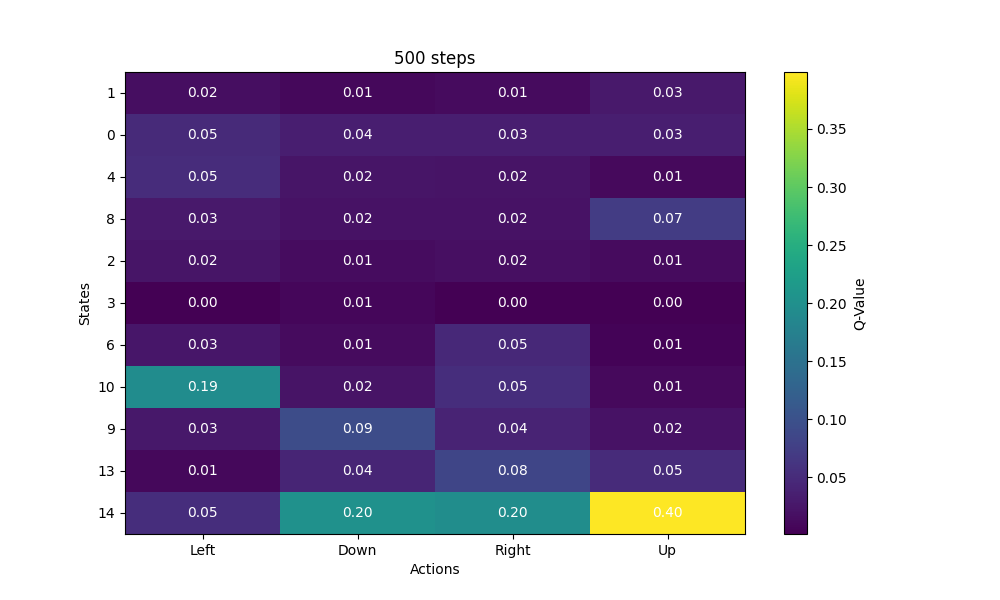
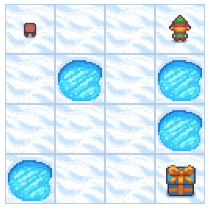


Figure 2: Q-value lookup table for different episodes



As can be seen in Figure.2, After 500 steps, most of the Q-values remained at 0. This makes sense because the agent starts with a random policy and performs many exploration steps. To receive a reward, the agent must reach the goal state. However, the probability of reaching the goal state by chance within a fixed number of actions is low. This explains why the Q-values did not update significantly at the beginning. As the number of steps increased, the Q-values began to stabilize. State 14 stands out because it is the only state that allows a direct transition to the goal state. Therefore, an agent positioned at state 14 has the highest probability of reaching the goal, which is why the maximum Q-values are observed in state 14. Similarly, the second-highest Q-value corresponds to state 13 moving right, as this action leads directly to state 14. Another observation is that the Q-value for state 3 is zero, likely because it is surrounded by many "holes" (negative or unrewarding states). The highest Q-value for state 3 is associated with moving left, which aligns with the agent avoiding these holes.

Figure 3: Frozen lake environment

Overall, the Q-value table aligns well with expectations, but the algorithm's performance is not optimal. The shortest possible path to complete the task is 7 steps, whereas the algorithm typically requires around 50 steps. This indicates that while the Q-learning algorithm worked to some extent, there is still room for improvement.

**Section 2 – Deep Q-learning:**

**Theoretical questions**

1. We sample experiences in random order to **break the correlation** between consecutive **state-action pairs** experienced by the agent. This reduces the variance caused by such correlations, leading to a more stable and effective training process.
2. Using an **older** **set of weights** to compute the targets improves the model by stabilizing training. It prevents the target values from changing **too rapidly**, which can cause instability and divergence in the learning process. Updating the target network every CCC steps allows the model to converge more reliably by providing a consistent reference for computing the loss.

**DQN agent:**

In this section, we were tasked with creating a DQN agent to operate in the Cart-Pole-V1 environment from Gymnasium. Each state was represented as a vector with four components: the cart's position, its velocity, the angle of the pole, and the angular velocity of the pole. We developed two DQN agents, one with 3 hidden layers and another with 5 hidden layers. For each agent, we tracked the loss at each step and the total reward for each episode.

**Agent-3 hidden layers**

The architecture of the DQN with 3 hidden layers consists of layer sizes: 64, 64, and 16. The following hyperparameters were used for this agent:

* **Learning rate**: 0.0005 (using the Adam optimizer)
* **Alpha**: 0.01
* **Batch size**: 256
* **Epsilon decay**: 0.9995
* **Discount factor**: 0.99
* **Initial epsilon**: 1
* **Final epsilon**: 0.005
* **Replay buffer capacity**: 10,000
* **C**: 100 (target network updated every 100 steps)
* **T**: 500 (up to 500 steps per episode)
* **Number of episodes**: 600 (training continues for up to 600 episodes or until the stopping criteria is met). The stopping criterion was achieved when the agent obtained an average reward of at least 475.0 over 100 consecutive episodes.

During hyperparameter tuning, we observed that setting the target network update frequency too low hindered the network's convergence, and we didn't see a clear improvement in reward values. Increasing the number of neurons in each layer led to longer training times due to the increased number of parameters that needed updating.

Another critical factor was the discount factor, which determines how much the agent prioritizes future rewards. A higher discount factor encourages the agent to consider long-term outcomes rather than focusing solely on immediate rewards. For instance, "saving" the agent from dropping the pole in the short term may not always be optimal, as the goal is to keep the pole balanced for as long as possible. After testing values like 0.95 and 0.9, we ultimately selected a discount factor of 0.99.

Additionally, we did not find a single hyperparameter that had the most significant impact on model performance. Instead, we found that a combination of adjustments to multiple hyperparameters contributed to a noticeable improvement in performance.

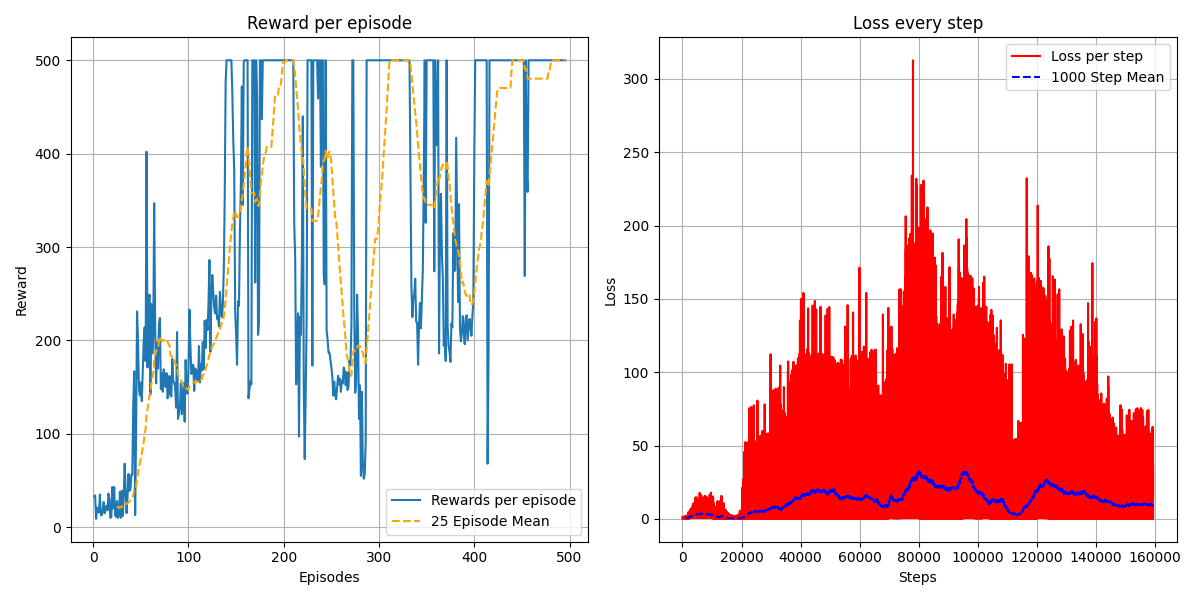
**Results:**

Figure 4: Results for the DQN agent with 3 hidden layers.

In most experiments, the DQN model successfully converged after a few hundred episodes, achieving an average reward exceeding 475. On average, the agent required approximately 350 episodes to reach this reward threshold. The left panel of the figure illustrates one of these runs, showing that the episodic rewards (blue line) stabilize after some variability, while the moving average (orange line) highlights that most agents met the stopping criteria between 350 and 400 episodes.

The right panel highlights an interesting trend in the loss curve. Initially, the loss decreases and converges, reflecting the model’s learning process. However, periodic spikes in the loss occur because the target values, used for loss computation, are updated every fixed number of episodes. These updates cause temporary increases in loss until the model adapts to the new targets, as shown by the red line (raw loss) and the blue dashed line (smoothed average). Overall, the figure demonstrates the agent’s ability to adapt and achieve stable performance after sufficient training.

**Agent-5 hidden layers:**

The architecture of the DQN with 3 hidden layers consists of layer sizes: 64, 64, 32, 32 and 16. The hyper-parameters that were used for this agent was the same as for the 3-hidden layer network.

**Results:**

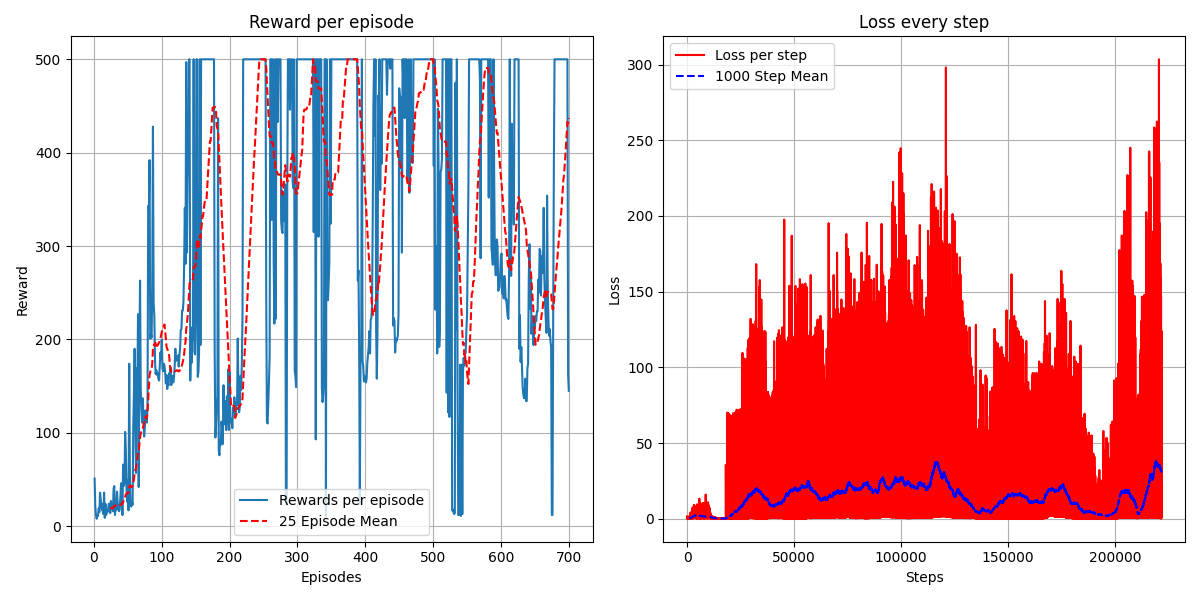


Figure 5: Results for the DQN agent with 5 hidden layers.

Besids not giving any sagnificant improvement, the extended network seem to perform even worse than the simpler network. Even several tries of tuning the parameters didn’t seem to get a better result. A possible explanation for that is the unneeded complexity for a rather simple problem. The smaller 3 – layer network, might be sufficient to approximate the Q-function effectively. Hance the additional layers of the extended network might introduce noise during the optimization or achieving overfitting (by memorize the data rather than generalize from it).

**Section 3 - Improved DQN:**

In this section, we had to make an improvement for the standard DQN algorithm and test its performance on the same environment as the DQN from section 2 (Cart-Pole-V1). For that part, we chose to go with the Double-DQN algorithm.

**Double – DQN algorithm:**

In this implementation, the algorithm uses **two separate Q-networks** to alternate the roles of **action selection** and **action evaluation**. The primary goal is still to reduce the overestimation bias of Q-values, but instead of using the same network for both tasks (as in DQN), Double DQN leverages two networks to perform these roles separately. The architecture of these two networks will be as same as the 3 hidden layers network from the DQN implematation (in section 2). The reason is to get better comperable results.

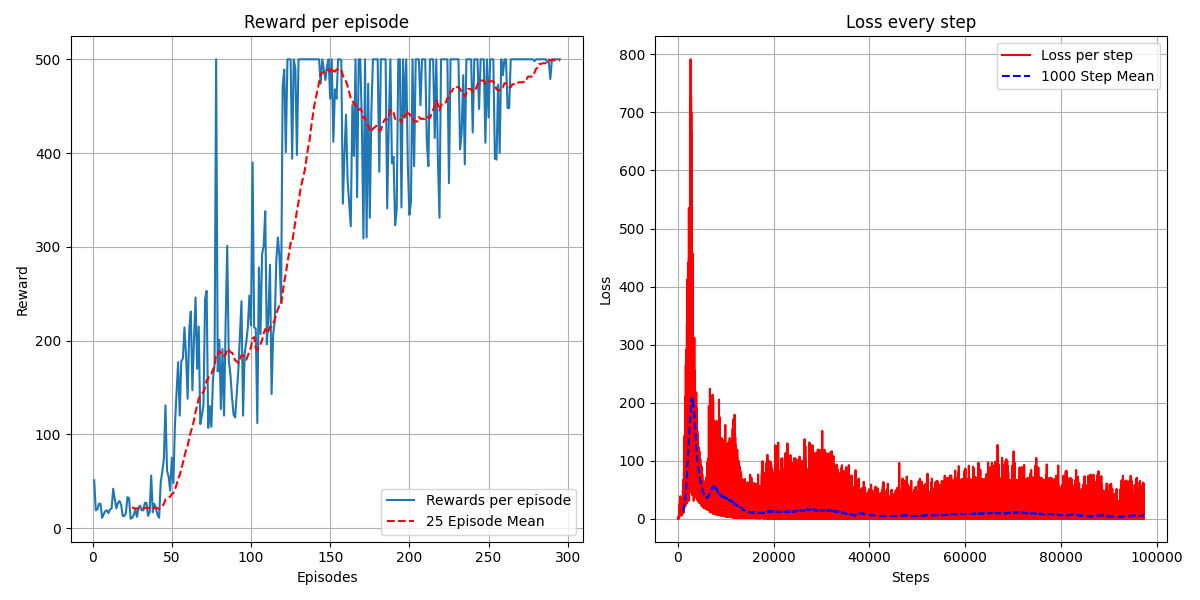
**Results:**

Figure 6: Results for Double-DQN

From these results, it’s easy to see a major improvement. On the left graph, it’s shown that the learning has achived the goal of obtaining an average reward of at least 475.0 over 100 consecutive episodes, **in less the 300 epsidos**. The average loss per step is much more stable with low values then with the DQN algorithm as well. Again, this is with same network architecture and hyper-parameters, which strongly implais on an improvement.

**Running the scripts:**

All the scripts for this assignment can be found either in the ZIP file that uploaded to the moodle, or in the following Git repo: <https://github.com/oz182/DRL_Assignments/tree/main>