Machine Learning Report

A Practical Study of Q-Learning & Deep Q-Network

Daidone Giuseppe 2122594 Sheibani Davood 2126056

Sapienza University of Rome

Summary

1 Problem Addressed

1.1 Problem Overview

The main objective of this study is to implement a Reinforcement Learning Agent following two approaches:

1. Tabular Q-Learning

2. Deep Q-Network (using one Neural Network to approximate the Q-function)

The Agent should be capable to learn how to solve a problem of a given environment.

1.2 Reinforcement Learning Environment

The agent studies a classic control problem from Gymnasium API of OpenAI. The environment selected to conduct this study is Cart Pole, derived from the cart-pole problem described by Barto et al. The problem is described as follows: *A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.*

1.2.1 Action Space

The action space is discrete and contains two values $\{0, 1\}$, indicating the direction in which the cart is pushed:

- 0: push to the left

- 1: push to the right

The velocity of the movement is not fixed and it depends on the angle the pole is pointing, since the center of gravity of the pole varies the amount of energy needed to move the cart underneath it.

1.2.2 Observation Space

The observation space is continuous and contains four values:

Num	Observation	Min	Max
0	Cart x-Position	-4.8	4.8
1	Cart Velocity	$-\infty$	∞
2	Pole Angle	~ -0.418 rad	~ 0.418 rad
3	Pole Angular Velocity	$-\infty$	∞

Episode terminates when one of the following events occur:

- Cart x-Position $\notin (-2.4, 2.4)$
- Pole Angle $\notin (-0.2095, 0.2095)$ rad
- Episode length > 500

1.2.3 Reward

The reward is always positive and it is assigned +1 for every step taken, including the termination step. The threshold (highest score) for the reward is 500, when this value is reached the episode terminates.

1.2.4 Starting State

All observations are assigned uniformly random value in $(-0.05, 0.05)$. Reward starts from 0.

2 Solution Adopted

2.1 Tools and Libraries

The practical implementation is performed using Python in a virtual environment created via PyCharm IDE. We also imported the following libraries:

- GYMNASIUM: to import the environment from Gymnasium
- NUMPY: to manipulate the arrays of action and state space and generate randomized policy
- TENSORFLOW and KERAS: to perform learning and build the Deep Neural Network
- MATPLOTLIB: to plot the performance evaluation
- Other libraries to support the computations and console logs

2.2 Development Environment

The virtual environment of PyCharm is used to install every library and dependencies needed to conduct the study. Furthermore, the environment is divided into two different directories having the following structure:

- Q-Learning
  - *q_learning.py*: performs the Tabular Q-Learning and computes the average return
  - *policies.py*: defines the possible policy to actuate (random policy, greedy policy, greedy policy with epsilon decay)
  - *main.py*: entry of the application
- Deep Q-Network
  - *replay_buffer.py*: implements the notion of experience memory of the agent

- *dqn.py*: defines a Deep Neural Network with train and evaluations methods
- *main.py*: entry of the application

2.3 Tabular Q-Learning Technique

Q-Learning is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. It does not require a model of the environment (hence "model-free"), and it can handle problems with stochastic transitions and rewards without requiring adaptations.
For any finite Markov Decision Process, Q-Learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state. Q-Learning can identify an optimal action-selection policy for any given finite Markov Decision Process, given infinite exploration time and a partly random policy. "Q" refers to the function that the algorithm computes, the expected rewards for an action taken in a given state.

2.3.1 Reinforcement Learning

Reinforcement Learning involves an agent, a set of states $\mathcal{S}$, and a set $\mathcal{A}$ of actions per state. By performing an action $a \in \mathcal{A}$, the agent transitions from state to state. Executing an action in a specific state provides the agent with a reward (a numerical score).

The goal of the agent is to maximize its total reward. It does this by adding the maximum reward attainable from future states to the reward for achieving its current state, effectively influencing the current action by the potential future reward. This potential reward is a weighted sum of expected values of the rewards of all future steps starting from the current state.

2.3.2 Algorithm

After Δt steps into the future the agent will decide some next step. The weight for this step is calculated as $\gamma^{\Delta t}$, where γ (the discount factor) is a number between 0

and 1 ($0 \leq \gamma \leq 1$). Assuming $\gamma < 1$, it has the effect of valuing rewards received earlier higher than those received later (reflecting the value of a "good start"). γ may also be interpreted as the probability to succeed (or survive) at every step Δt.

The algorithm, therefore, has a function that calculates the quality of a state–action combination:

$$Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$$

Before learning begins, Q is initialized to a possibly arbitrary fixed value (chosen by the programmer). Then, at each time t the agent selects an action A_t, observes a reward R_{t+1}, enters a new state S_{t+1} (that may depend on both the previous state S_t and the selected action), and Q is updated. The core of the algorithm is a Bellman equation as a simple value iteration update, using the weighted average of the current value and the new information:

$$Q^{\text{new}}(S_t, A_t) \leftarrow (1 - \alpha) \cdot Q(S_t, A_t) + \alpha \cdot (R_{t+1} + \gamma \cdot \max_a Q(S_{t+1}, a))$$

where R_{t+1} is the reward received when moving from the state S_t to the state S_{t+1}, and α is the learning rate ($0 < \alpha \leq 1$). Note that $Q^{\text{new}}(S_t, A_t)$ is the sum of three factors:

- $(1 - \alpha) \cdot Q(S_t, A_t)$: the current value weighted by one minus the learning rate
- $\alpha \cdot R_{t+1}$: the reward to obtain if action A_t is taken when in state S_t weighted by learning rate
- $\alpha \cdot \gamma \max_a Q(S_{t+1}, a)$: the maximum reward that can be obtained from state S_{t+1} weighted by learning rate and discount factor

An episode of the algorithm ends when state S_{t+1} is a final or terminal state. However, Q-Learning can also learn in non-episodic tasks (as a result of the property of convergent infinite series). If the discount factor is lower than 1, the action values are finite even if the problem can contain infinite loops.

For all final states s_f, $Q(s_f, a)$ is never updated, but is set to the reward value r observed for state s_f. In most cases, $Q(s_f, a)$ can be taken to equal zero.

2.4 Deep Q-Network Technique

The Deep Q-Network (DQN) technique is employed to train an agent to perform optimally in a given environment. This approach utilizes a neural network to approximate the Q-Values, which represent the expected cumulative reward for taking a given action in a given state. The main components and processes involved in the DQN technique are as follows.

2.4.1 Replay Buffer

A data structure used to store past experiences (state, action, reward, next state, done). This buffer allows the agent to learn from a diverse set of experiences and break the correlation between consecutive experiences by sampling random mini-batches for training.

2.4.2 Epsilon-Greedy Policy

A strategy for balancing exploration and exploitation. The agent selects a random action with probability epsilon (exploration) and the action with the highest Q-Value with probability $1-\epsilon$ (exploitation). Epsilon decays over time to reduce exploration as the agent learns.

2.4.3 Experience Replay

A technique where past experiences are sampled from the replay buffer to update the Q-Network. This approach improves learning efficiency and stability by allowing the agent to learn from a wider range of experiences.

2.4.4 Training Process

The training process can be broken down into the following steps:

1. Initialize Environment and State: the agent starts with an initial state provided by the environment.

2. Action Selection: the agent selects an action based on the epsilon-greedy policy.

3. Interact with Environment: the selected action is executed in the environment, leading to a new state and reward.

4. Store Experience: the (state, action, reward, next state, done) tuple is stored in the replay buffer.

5. Update Q-Network: experiences are sampled from the replay buffer, and the Q-Network is updated using these experiences.

    i) Perform a forward pass to get predicted Q-Values

    ii) Compute the target Q-Values

    iii) Calculate the loss using the Mean Squared Error (MSE) loss function

    iv) Perform backpropagation to update the Q-Network weights

6. Update Target Model: periodically, the weights of the Q-Network are copied to the target model to stabilize training.

3 Architecture of the Implementation

3.1 Q-Learning Architecture

The core idea of Q-Learning is to learn a Q-function, which is a mapping from state-action pairs to expected future rewards. This function is typically stored in a Q-Table in Tabular Q-Learning.

3.1.1 Q-Table

A table where each entry $Q(s, a)$ represents the expected future reward of taking action a in state s. The table is initialized with arbitrary values (often zeros) and is updated iteratively as the agent learns from its interactions with the environment.

3.1.2 Parameters Used in Q-Learning

- **Learning Rate**: The learning rate or step size determines to what extent newly acquired information overrides old information. A factor of 0 makes the agent learn nothing (exclusively exploiting prior knowledge), while a factor of 1 makes the agent consider only the most recent information (ignoring prior knowledge to explore possibilities). In practice, often a constant learning rate is used, such as $\alpha_t = 0.1$ for all t.

- **Discount Factor**: The discount factor γ determines the importance of future rewards. It is a value between 0 and 1. A discount factor of 0 makes the agent consider only immediate rewards, while a discount factor closer to 1 makes the agent strive for long-term rewards. The choice of γ affects the agent's foresight: a higher γ makes the agent more far-sighted, considering future rewards more heavily in its decisions.

- **Initial Conditions (Q0)**: The initial conditions for the Q-Values can significantly impact the exploration strategy of the agent. We initialized all Q-Values to 0, providing a neutral starting point. This choice encourages exploration without biasing the agent towards any particular state-action pairs initially.

- **Number of Steps**: The number of steps is the number of learning rounds to build the Q-Table. In our model, we set $maxStep = 2 \cdot 10^5$.

- **Epsilon Decay**: The epsilon decay technique defines which is the function that describes the decreasing value of ε, which is the probability to take a random action (exploration) instead of an action based on learning situation (exploitation). We applied the exponential epsilon decay technique:

$$\varepsilon = \varepsilon_{min} + \varepsilon_0 \cdot e^{-decayRate \cdot step}$$

  where the epsilon is decreasing from $\varepsilon_0 = 1$ to $\varepsilon_{min} = 0$, the step is increasing linearly from 0 to $maxStep - 1$ and $decayRate = 10^{-5}$.

- **Discretization bins**: Since the observation space for the studied environment is continuous we discretized it, partitioning equally with a number of samples $bins = 40$.

3.2 DQN Architecture

The DQN was implemented using TensorFlow and Keras libraries and the architecture consists of a Feedforward Neural Network. In addition, we used a Target Neural Network with the same architecture of the main DQN to provide stable reference point for computing the values during the training. It helps to prevent the oscillations and divergence in the Q-Values that can occur if the main DQN updates too frequently. The target network is updated less frequently by copying the weights from main DQN (every 10 episodes).

3.2.1 Neural Network

The neural network for the DQN is composed by:

- An input layer matching the state size.

- Two hidden layers, each with 64 neurons and ReLU activation functions.

- An output layer matching the action size with a linear activation function.

3.2.2 Parameters Used in DQN

The implementation utilized the following parameters:

- **Discount Factor**: $\gamma = 0.99$
- **Learning Rate**: $\alpha = 0.001$
- **Batch Size**: 64
- **Memory Size**: 20,000
- **Episodes**: 50
- **Exploration Rate**: $\varepsilon = 0.1$

3.2.3 Loss Function and Backpropagation

The loss function and backpropagation are critical components of training the Q-Network.

- **Loss Function**: The loss function used is the Mean Squared Error (MSE) between the predicted Q-Values and the target Q-Values. The target Q-Value is calculated as the reward plus the discounted maximum Q-Value of the next state.

$$\text{Loss} = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - Q(s_i, a_i \mid \theta) \right)^2$$

  where y_i is the target Q-Value and $Q(s_i, a_i \mid \theta)$ is the predicted Q-Value from the Q-Network.

- **Backpropagation**: Backpropagation is used to minimize the loss function by updating the weights of the Q-Network. The Adam optimizer is employed to adjust the weights based on the gradients of the loss function.

3.3 Computational Power Influence

It is important to highlight that the computational power available to conduct the tests might influence the experimental results found in this study. Although the

numerical values can be affected by the available computational power during the running (not always constant), conducting the same tests on different machines have led to the same conclusions. Moreover, since the experiments may require a large amount of data to be stored (e.g., the table of q-learning, discretization of the environment), replicating these conditions on a less powerful machine can cause overflow yielding the running to be terminated.

4 Experimental Results

In order to evaluate how well the two agents behave, we have plotted the performance evaluation. The two plots have different (but affine) structure:

- Q-Learning
  - Reward over Episodes: visualize the reward for each episode in learning
  - Epsilon Decay over Steps: visualize the shape of the function that defines the epsilon decay technique
  - Average Q-Value over Steps: visualize the average Q-Value for each step
- DQN
  - Reward over Episodes: visualize the reward for each episode in learning and the trend (regression) line

4.1 Q-Learning Performance Evaluation

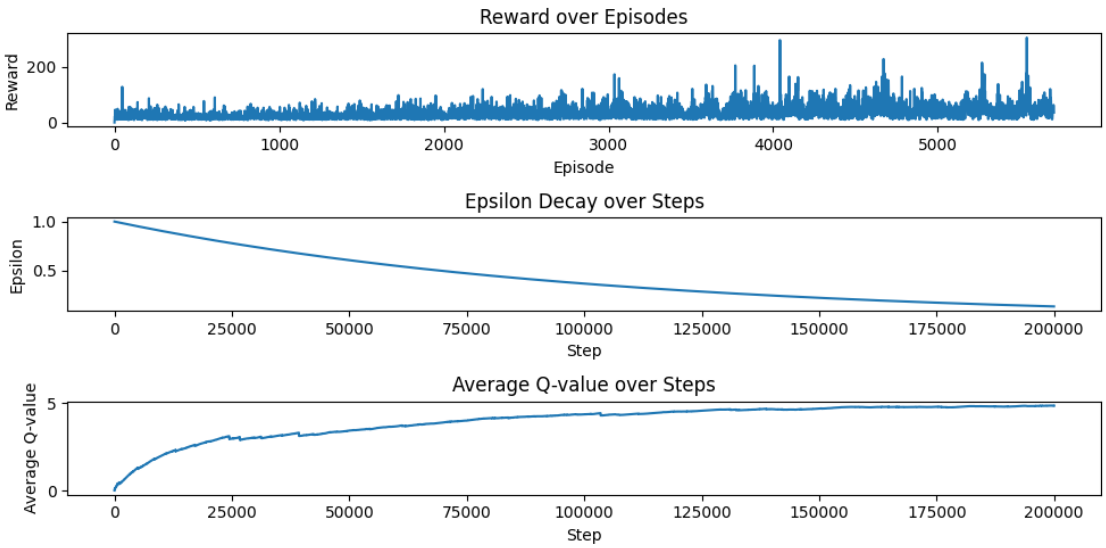


Figure 1: Q-Learning Performance Evaluation

4.2 DQN Performance Evaluation

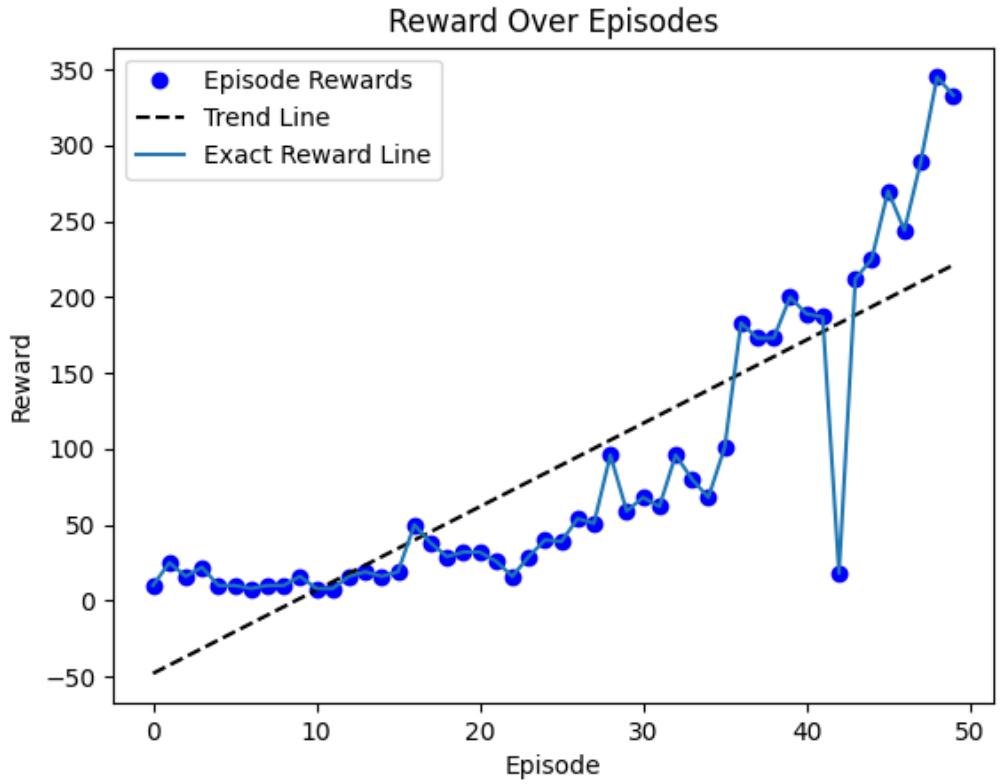


Figure 2: Deep Q-Network Performance Evaluation

4.3 Average Reward Comparison

The average reward returned by the agents are the following:

	Average Reward
Random Policy	9.79
Q-Learning	19.83
DQN	311.4

As we expected, the DQN model has performed considerably better than the Q-Learning model.

5 Further Tests

We wanted to conduct more tests for both Q-Learning and DQN by increasing or decreasing some parameters in order to study which was their impact, how the plot shape changed and if some parameters combination could have led to a better solution in terms of reward. In the following tables of comparison, the row highlighted represents our main model while the marked cell represents the best value along the analyzed models. We will refer as "our model" as the main models presented above in this study.

5.1 Reward over Discretization Bins

In order to evaluate whether the discretization bins for discretizing the continuous environment are the optimum, we tested different combinations by augmenting or reducing the number of bins. In addition, we also changed the number of steps accordingly.

In the first test conducted, we divided by half both the number of discretization bins and steps. The values returned are Average Reward = 14.95 and Single Episode Reward = 18.98. The corresponding plot is the following:

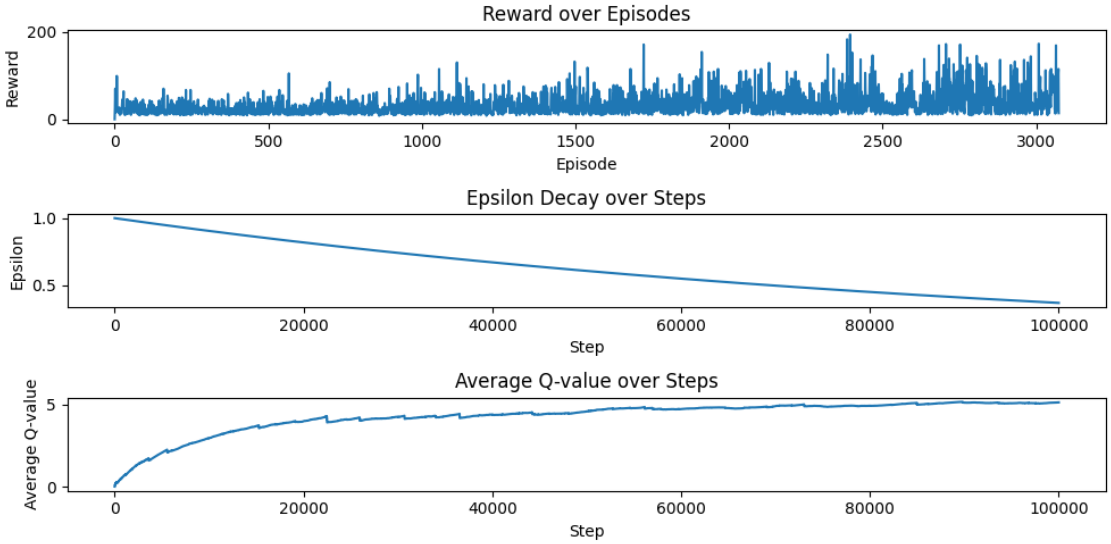


Figure 3: Q-Learning Performance Evaluation 20 bins 100k steps

Since the average reward in the previous test was much less than our model,

the value for number of steps was reset to 200k, but maintaining 20 discretization bins. The values returned are Average Reward = 17.56 and Single Episode Reward = 19.99. The corresponding plot is the following:

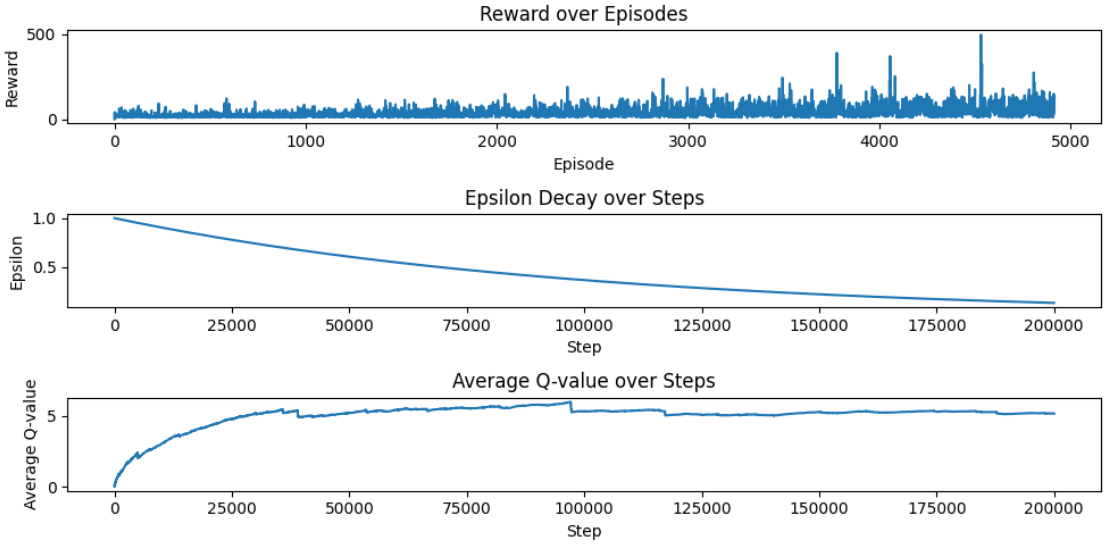


Figure 4: Q-Learning Performance Evaluation 20 bins 200k steps

The average reward remains less than our model, but the single episode reward seems slightly better. In the last test we doubled both the number of discretization bins and steps. The values returned are Average Reward = 17.18 and Single Episode Reward = 17.74. The corresponding plot is the following:

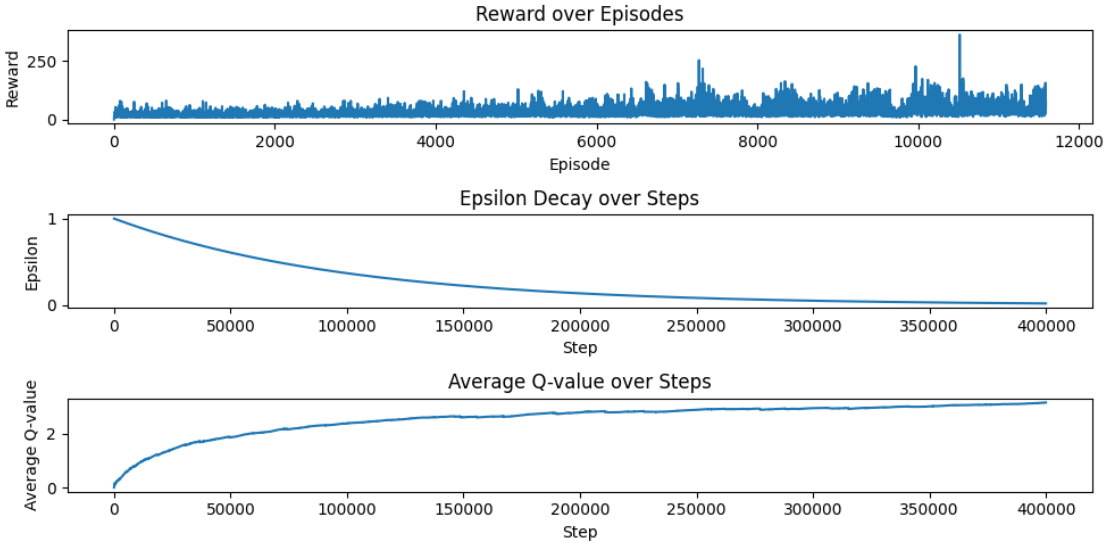


Figure 5: Q-Learning Performance Evaluation 80 bins 400k steps

Comparison between the different parameters combination:

Bins & Steps in Q-L	**Average Reward**	**Single Episode Reward**
20 bins, 100k steps	14.95	18.98
20 bins, 200k steps	17.56	19.99
40 bins, 200k steps	19.83	19.93
80 bins, 400k steps	17.18	17.74

Even if the combination of 20 bins and 200k steps is slightly better for single episode reward, the average reward of our model is much better among the others.

5.2 Impact of Different Epsilon Decay Techniques

In our Q-Learning model we used the exponential epsilon decay technique, where epsilon value was computed as:

$$\varepsilon = \varepsilon_{min} + \varepsilon_0 \cdot e^{-decayRate \cdot step}$$

where:

$$\varepsilon \in [0, 1], \varepsilon_{min} = 0, \varepsilon_0 = 1$$
$$decayRate = 10^{-5}$$
$$step \in [0, maxStep - 1], maxStep = 2 \cdot 10^5$$

We tested the Q-Learning model with linear epsilon decay technique, where epsilon value was computed as:

$$\varepsilon = \frac{maxStep - step}{maxStep}$$

Performance evaluation plot with linear epsilon decay:

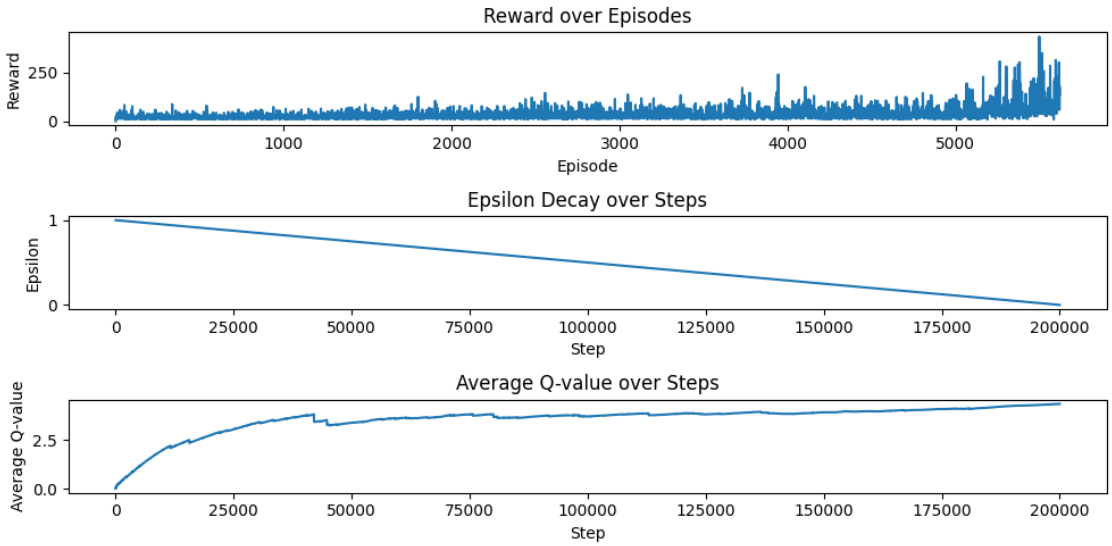


Figure 6: Q-Learning Performance Evaluation with Linear Epsilon Decay

Comparison of average reward and single episode reward with the two different epsilon decay technique:

Epsilon Decay Technique	**Average Reward**	**Single Episode Reward**
Linear	19.33	19.67
Exponential	19.83	19.93

Exponential decay performed better with respect to linear decay. Although the difference of rewards between the two techniques appears to be relatively small, increasing the number of steps would increase this difference proving that exponential epsilon decay is better for this model. This phenomena happens because, for smaller steps, the exponential epsilon decay function approximates the linear epsilon decay function. In fact, this limit holds:

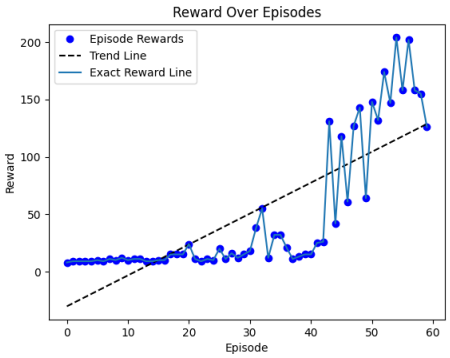
$$\lim_{steps \to 0} \varepsilon_{exp} = \varepsilon_{lin} = 1$$

We can conclude that at some point the agent should prefer with highest probability exploitation over exploration. Estimating this point as the half of the step graph as in the linear epsilon decay case does not perform better in learning.

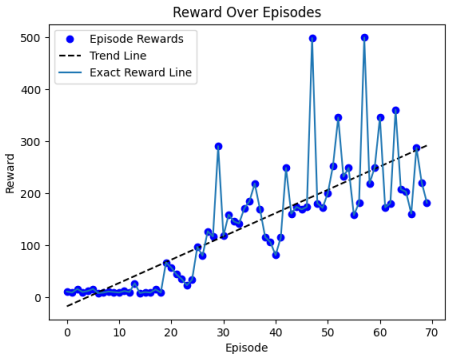
5.3 Number of Learning Episodes for DQN

We wanted to test if for higher number of episodes we could have got a better average reward for the Deep Q-Network.

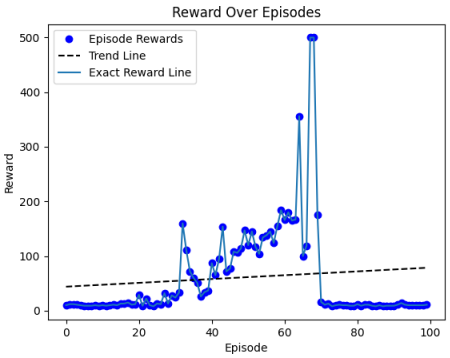
Performance evaluation plots for different episodes number:



(a) DQN Performance Evaluation 60 episodes



(b) DQN Performance Evaluation 70 episodes



(c) DQN Performance Evaluation 100 episodes

The plot of DQN with 100 episodes is showing an immediate decay in performances around episode $\sim$ 70. At first, the hypothesis for this behaviour was the Overfitting phenomena but some researches has shown that this shape suggests a problem called Catastrophic (or Interference) Forgetting. This problem occurs when the agent learns for too many episodes, causing it to forget about the learned policy. The performances for this model results as a random policy.

Comparison of rewards for different episodes:

Episodes in DQN	**Average Reward**
50 episodes	311.4
60 episodes	212.9
70 episodes	252.1
100 episodes	9.9

Our DQN model with 50 episodes has performed better among the others, so we can conclude that 50 is the most appropriate number of learning episodes for this settings.