Project

Machine Learning

Facoltà di ingegneria dell'informazione, informatica e statistica
Engineering in Computer Science
Sapienza, Rome, Italy

Andrea Panceri 1884749, Francesco Sudoso 1808353

Academic Year 2022/2023

Contents

List of Figures

1 Reinforcement Learning

Reinforcement learning is a method for solving problems involving continuous decision-making. It is a method employed for discovering an agent's behavioral policy that enables the agent to perceive the current state of the environment and choose the action that maximizes the cumulative reward. The agent interacts with the environment and executes multiple actions in order to determine the optimal policy by obtaining reward feedback on each action. Two common approaches to reinforcement learning are the **Q-table** method and **Q-function approximation**.

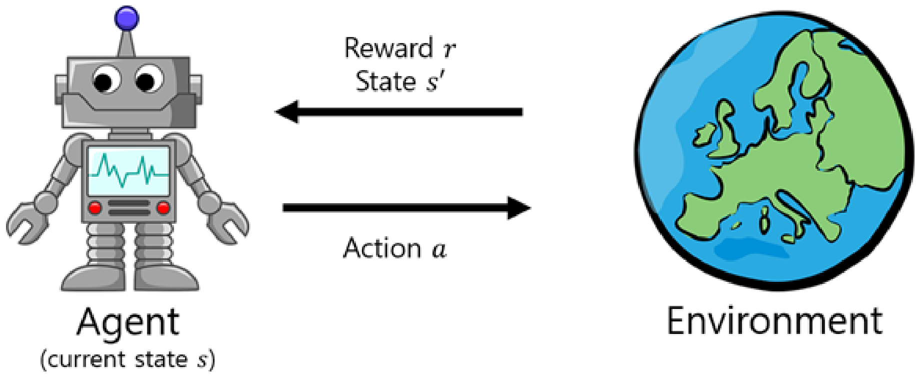


Figure 1: Reinforcement Learning

The Q-table method is a straightforward and tabular representation of the Q-values for each state-action pair in the environment. The **Q-value** represents the expected cumulative reward an agent can obtain by taking a particular action in a specific state. Initially, the Q-table is empty, and the agent explores the environment to fill it up.
During training, the agent takes actions based on an exploration-exploitation strategy, such as epsilon-greedy. It chooses either the action with the highest Q-value (**exploitation**) or a random action (**exploration**) with a certain probability. After each action, the agent updates the Q-value in the Q-table based on the observed reward and the maximum Q-value of the next state. This update follows the Bellman equation, which aims to optimize the Q-values iteratively and Convergence is guaranteed only if every possible state-action pair is visited infinitely often.
However, the Q-table method has limitations when the environment becomes large or continuous, as it requires storing and updating values for every state-action pair. This leads us to Q-function approximation method.
Q-function approximation is an approach that uses a function, typically a parametric model, to approximate the Q-values instead of storing them in a tabular form. This function takes the state-action pair as input and outputs an estimate of the Q-value. One common approach for Q-function approximation is using a neural network as the function approximator, resulting in a technique called deep Q-networks (**DQNs**).
In Q-function approximation, the agent still follows a similar exploration-exploitation strategy as in the Q-table method, but instead of updating individual Q-values, it trains the neural network to approximate the Q-values based on a loss function. The **loss function** is defined using the Bellman equation and the observed rewards.
The advantage of Q-function approximation is that it can handle large and continuous state spaces more efficiently. However, it introduces challenges such as convergence issues and the need for careful selection of network architecture and training strategies to ensure stabil-

ity and effective learning.
Overall, both the Q-table method and Q-function approximation are techniques used in reinforcement learning to enable agents to learn and make decisions in an environment. The Q-table method is suitable for small and discrete environments, while Q-function approximation is more suitable for larger and continuous environments.

2 Taxi-Environment

The **Taxi environment** is Toy Text environments included in Gymnasium that is often used to demonstrate reinforcement learning algorithms. It simulates a simplified scenario where a taxi operates in a grid world. Let's go through the various components of the Taxi-environment:

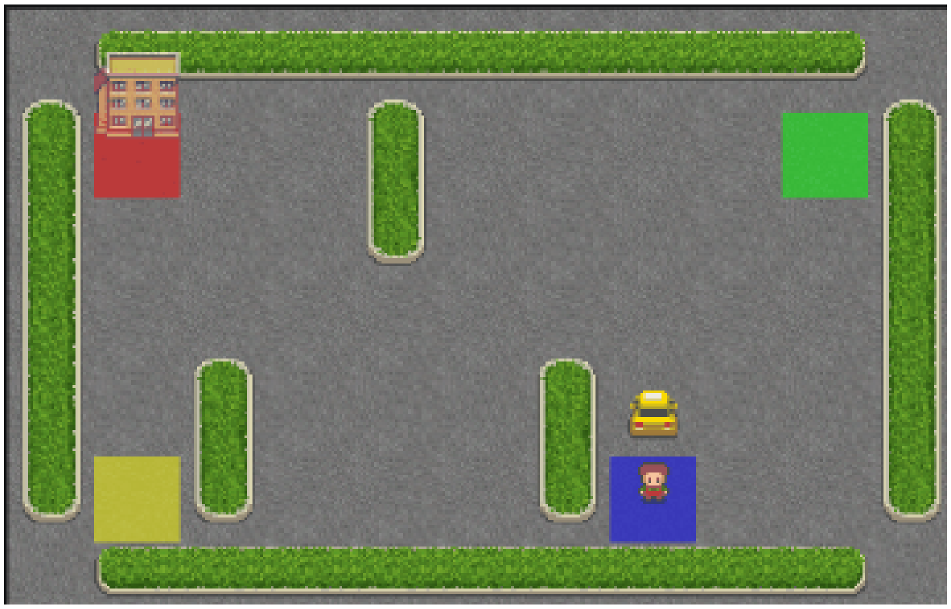


Figure 2: Taxi Environment

- States:
    - **Grid World:** The environment consists of a grid of squares. Each square represents a location where the taxi and passengers can be. There are 500 discrete states since there are 25 taxi positions, 5 possible locations of the passenger (including the case when the passenger is in the taxi), and 4 destination locations.
    - **Taxi:** The taxi can be in one of the grid squares. There are four possible locations for the taxi to start.
    - **Passengers:** There are four passengers denoted by different letters: R (red), G (green), Y (yellow), and B (blue). The passengers can be either inside the taxi or at one of the specific locations (R, G, Y, or B).
    - **Destinations:** There are four destinations corresponding to each passenger's letter. The taxi's goal is to pick up a passenger and deliver them to their corresponding destination.
- **Actions:** the taxi can perform six possible discrete actions

  - Move south.
  - Move north.
  - Move east.
  - Move west.
  - Pick up a passenger.
  - Drop off a passenger.

- **Rewards:**
  - For each time step taken, there is a penalty of -1 to encourage the taxi to find the shortest path.
  - When the taxi successfully drops off a passenger at the correct destination, it receives a reward of +20.
  - If the taxi tries to pick up or drop off a passenger at the wrong location, it receives a penalty of -10.

- **Settings:**
  - The Taxi environment is deterministic, meaning the outcome of an action is always the same.
  - The state space is discrete and has a fixed size.
  - The action space is discrete with a fixed number of possible actions.
  - The environment is episodic, meaning each episode starts with a random initial state and ends when the taxi drops off a passenger at the correct destination.
  - The environment resets at the end of each episode, and the taxi is placed in a random location.

- **Goal:**
  - The goal of the Taxi environment is for the taxi to learn an optimal policy to pick up passengers from their current locations and transport them to their corresponding destinations efficiently. The taxi needs to navigate the grid world, avoid incorrect pickups or drop-offs, and minimize the number of steps taken to earn the maximum cumulative reward.

Reinforcement learning algorithms can be applied to train an agent to solve the Taxi environment by iteratively updating the Q-values or using Q-function approximation to find an optimal policy that maximizes the cumulative rewards.

3 Solution Adopted

In the following section, we will describe the data structures and algorithms we adopted to successfully implement the learning.

3.1 Learning Algorithm (S.A.R.S.A)

The **SARSA** algorithm is a reinforcement learning algorithm that stands for *"State-Action-Reward-State-Action."* It is an on-policy algorithm, meaning it learns the optimal policy while following the current policy. It is often used to approximate the optimal value of the Q-function for a given environment.

The algorithm consists of the following steps:

- Observe the current state **S**.
- Choose an action, **A**, based on the epsilon-greedy policy
- Execute the chosen action, **A**, and observe the next state, **S'**, and the reward, **R**.
- Choose the next action, **A'**, based on the policy and the next state, **S'**.
- Update the Q-value for the current state-action pair using the Q-Function (in our case **non-deterministic Q-Function**)

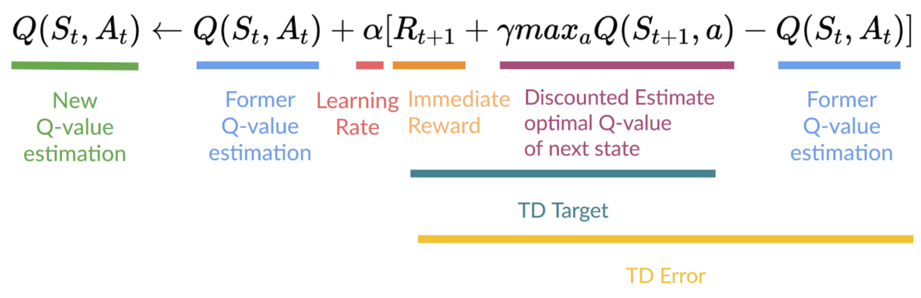

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$
New Q-value estimation
Former Q-value estimation
Learning Rate
Immediate Reward
Discounted Estimate optimal Q-value of next state
Former Q-value estimation
TD Target
TD Error

Figure 3: Q-Learning

We repeat these steps until a maximum number of steps is reached or the *done* value is *True*. The SARSA algorithm updates the Q-values based on the current action and the next action chosen following the same policy. This characteristic makes it an on-policy algorithm, as it learns and improves the policy it follows during training. SARSA is well-suited for problems where exploration is important, as it explicitly considers the exploration action taken at each step.

3.2 Replay Buffer

In reinforcement learning, a **replay buffer** is a data structure that is commonly used algorithms of Q-learning. The replay buffer stores the experiences variables *(state, action, reward, next state)* observed during the agent's interaction with the environment.
We made use of the replay buffer in the following way:

- Initialize an empty replay buffer with a fixed capacity.

- During the agent's interaction with the environment, we store the transition **(state, action, reward, next state)** in the replay buffer.
- Finally, when we want to update of the Q-function, we sample **a batch of transitions** from the replay buffer and we use it for updating the Q-Function.

The replay buffer helps to break the temporal correlation between consecutive transitions, making the learning process more stable and efficient. By randomly sampling transitions from the replay buffer, the agent can learn from a diverse set of experiences, reducing the bias that can arise from sequential correlations. The replay buffer also enables the agent to learn from past experiences multiple times, as transitions are stored and reused during the training process. For this reason, it is useful **for improving the learning stability and efficiency**.

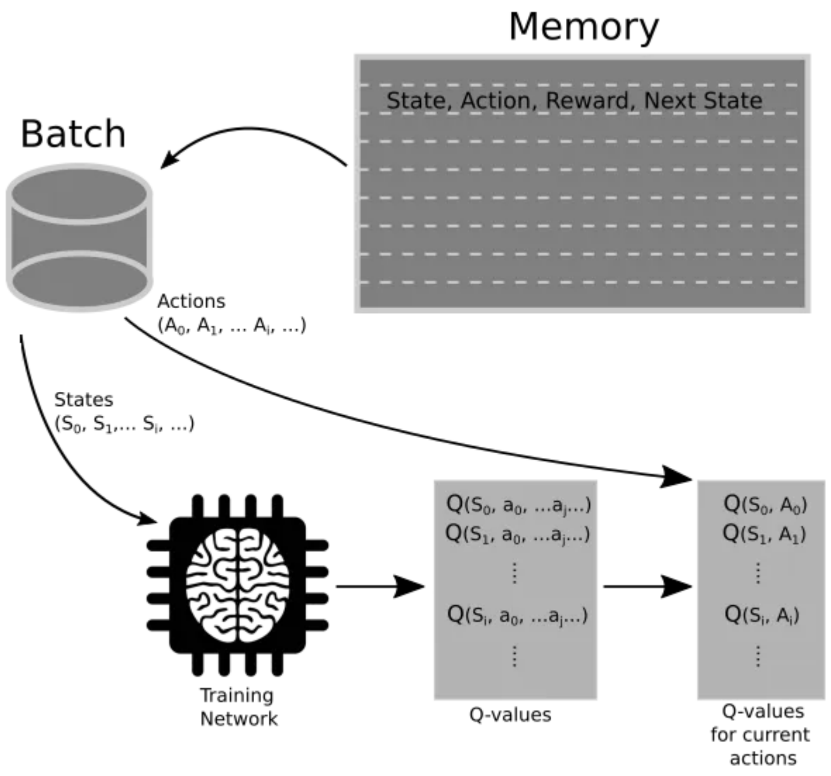

Memory
State, Action, Reward, Next State
Batch
Actions
$(A_0, A_1, \ldots A_i, \ldots)$
States
$(S_0, S_1, \ldots S_i, \ldots)$
$Q(S_0, a_0, \ldots a_j \ldots)$
$Q(S_1, a_0, \ldots a_j \ldots)$
$Q(S_i, a_0, \ldots a_j \ldots)$
$Q(S_0, A_0)$
$Q(S_1, A_1)$
$Q(S_i, A_i)$
Training Network
Q-values
Q-values for current actions

Figure 4: Replay Buffer

3.3 Q-Table

As we said before, in reinforcement learning, Q-learning is an algorithm used to find the optimal action-selection policy using a Q-function. The Q table helps us to find the best action for each state. It's like a simple lookup table where the columns are the actions and the rows are the states, and helps to maximize the expected reward by selecting the best of all possible actions. The Q-learning algorithm works as follows:

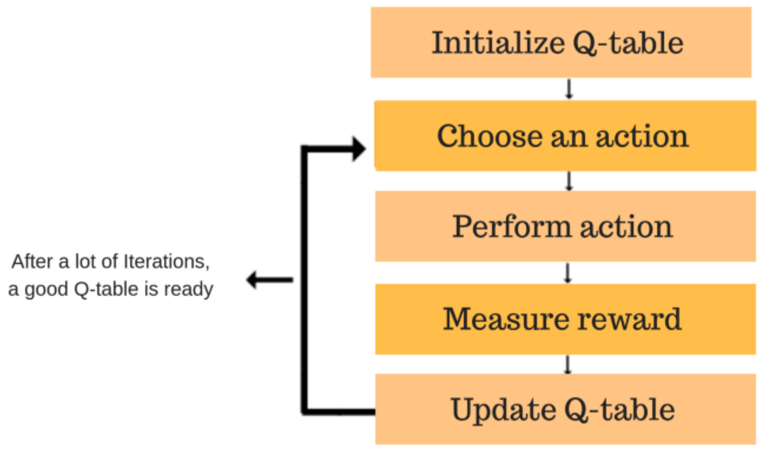


Figure 5: Q-learning algorithm process

In our solution, we created two functions, namely the **Qlearn** function to do the learning and the **Rollouts** function to test the correctness of the algorithm. Regarding the first function, we first defined a set of constants:

```
EPS_MAX = 1
EPS_MIN = 0.01
decay_epsilon = 0.001
epsilon = EPS_MAX
WRONG_ACTION = -10
cumulative_reward = 0
number_of_wrong_actions = 0
```

In particular, the **epsilon** refers to the probability of choosing to explore. **EPS_MAX** is the maximum (starting) value of the epsilon, that means, initially only exploration. **EPS_MIN** is the minimum value that the epsilon can reach. **decay_epsilon** corresponds to the rate of decay of the epsilon, such that from the maximum value it reaches the minimum value after a certain number of episodes. As we will see later, we adopted an exponential decay of the epsilon. The last three variables, such as **wrong_action**, **cumulative_reward** and **number_of_wrong_actions** will be used for evaluating the performance of the learning.

In addition, we initialize a Qtable with all values set to 0 using a lambda function. In particular, we have 500 rows(one for each state) and 6 columns(one for each possible action):

```
Q = defaultdict(lambda: np.zeros(enviroment.action_space.n))
```

After that, we define a policy for selecting an action as follows:

```
policy = EpsGreedyPolicy(Q)
```

where **EpsGreedyPolicy** is a class defined as follows:

```
class EpsGreedyPolicy:
    def __init__(self, Q):
        self.Q = Q
        self.n_actions = len(Q[0])
    def __call__(self, obs, epsilon: float) -> int:
        greedy = random.uniform(0,1) > epsilon
        if greedy:
            return np.argmax(self.Q[obs])
        else:
            return random.randint(0, self.n_actions - 1)
```

The EpsilonGreedyPolicy is a simple method to balance exploration and exploitation by choosing between exploration and exploitation randomly. It selects the action with the highest Q-value with probability 1-epsilon and selects a random action with probability epsilon. In case of exploitation, the action is selected from the QTable.

The most important part of our solution is how we do the learning and the how we update the table:

```
for episode in range(number_of_episodes):

    state, _ = enviroment.reset()
    done = False

    #Iterate until episode is Done
    for step in range(max_steps_per_episode):

        #WRONG_ACTION of passenger
        if done:
            break

        # Select action using Epsilon Greedy police
        action = policy(state, epsilon)
        next_state, reward, done, _ , _ = enviroment.step(action)

        #Illegal WRONG_ACTION
        if reward == WRONG_ACTION:
            number_of_wrong_actions += 1

        #Updating of Q table and updating of state
        Q[state][action] = (1- alpha) * Q[state][action] + alpha * (reward + gamma * np.max(Q[next_state]))
        state = next_state

        cumulative_reward += reward

    cumulative_reward = 0
    number_of_wrong_actions = 0

    # Exploration rate esponential decay
    epsilon = EPS_MIN  + (EPS_MAX - EPS_MIN) * np.exp(-decay_epsilon*episode)
```

In this part of the algorithm we perform for each episode, a number of action or steps that is at most **max_steps_per_episode**, because in this way the learning is faster.
At the beginning of each episode, we reset the environment for restarting the game. Then, for each step, we select an action through our policy and we execute it, saving the next state, the reward and a boolean value that corresponds to the termination of the game or not. After, only for evaluation purpose, we check if the selected action gave us -10 as reward, and in that case we record that a wrong action was selected.

The next step of the algorithm is to update the Qtable: we access the table cell corresponding to the state (as row) and action (as column) and we update the corresponding value with the deterministic Qfunction value. Finally, we set the new state as the current state and add the current reward to the cumulative reward.

In the next iteration, we check first of all if the **done** variable is True. In this case, we break the cycle, reset the cumulative reward and the number of wrong actions variables, we decrease exponentially the exploration rate and we start a new episode. The algorithm will continue in this manner until the number of episodes passed as a parameter to the Qlearn function are executed. At the end of the execution, if we have passed a good amount of episodes as variable, we will have a Qtable populated with the best values for each state and action.

In order to evaluate whether the learning was successful, we defined a function called **Rollout** that is very similar to the previous function, but in this case we do not perform any update of the table, but only exploit it for the selection of the best action in a given state. For exploiting the Qtable we also defined a new class policy called **GreedyPolicy** that given the Qtable and a state, it return the best action to perform.

```
class GreedyPolicy:
    def __init__(self, Q):
        self.Q = Q
    def __call__(self, obs) -> int:
        return np.argmax(self.Q[obs])

def Rollouts(enviroment, policy, number_of_episodes, render):
    total_reward = 0
    done = False
    state, _ = enviroment.reset()

    episodes = 0
    if render:
        enviroment.render()

    # Iterate steps
    while True:
        if done:
            if render:
                print("Episode " + str(episodes) + " terminated.")
            state, _ = enviroment.reset()
            episodes += 1
            if episodes >= number_of_episodes:
                break

        # Select action
        action = policy(state)
        state, reward, done, _ , _ = enviroment.step(action)

        total_reward += reward

        if render:
            enviroment.render()

    return total_reward / number_of_episodes
```

3.3.1 Qtable performance

To maximize learning, the fine-tuning of the parameters used to calculate Q-function, such as **alpha**, **gamma** and **decay_rate** is necessary. Alpha is the learning rate and determines the weight given to new information compared to existing knowledge when updating the Q-values. A higher learning rate means that new information has a greater impact, instead a lower learning rate assigns more weight to existing knowledge, resulting in slower updates. Gamma is the discount factor, which determines how much importance we give to future rewards. A high value of gamma will result in long-term rewards being considered more than short-term rewards. We have tried different combination of these parameters in order to evaluate which is the best configuration. In particular, we tested the following configurations:

- $\alpha = 0.01, \gamma = 0.99, decay\_rate = 0.001$
- $\alpha = 0.1, \gamma = 0.99, decay\_rate = 0.001$
- $\alpha = 0.1, \gamma = 0.99, decay\_rate = 0.01$
- $\alpha = 0.5, \gamma = 0.99, decay\_rate = 0.001$

For each test, we plotted the cumulative reward, the number of wrong actions, and the epsilon decay. From the following charts we can see that the best parameter configuration is the last one, that is $\alpha = 0.5, \gamma = 0.99, decay\_rate = 0.001$.
This is because it takes **1000 episodes** to achieve the best cumulative reward, compared to 1500 for the third configuration and 2000 for the second configuration. Same analysis applies to the number of wrong actions.
In addition, the last configuration seems to be the most stable in terms of reward, as we can see fewer negative spikes in the charts than the other configurations.

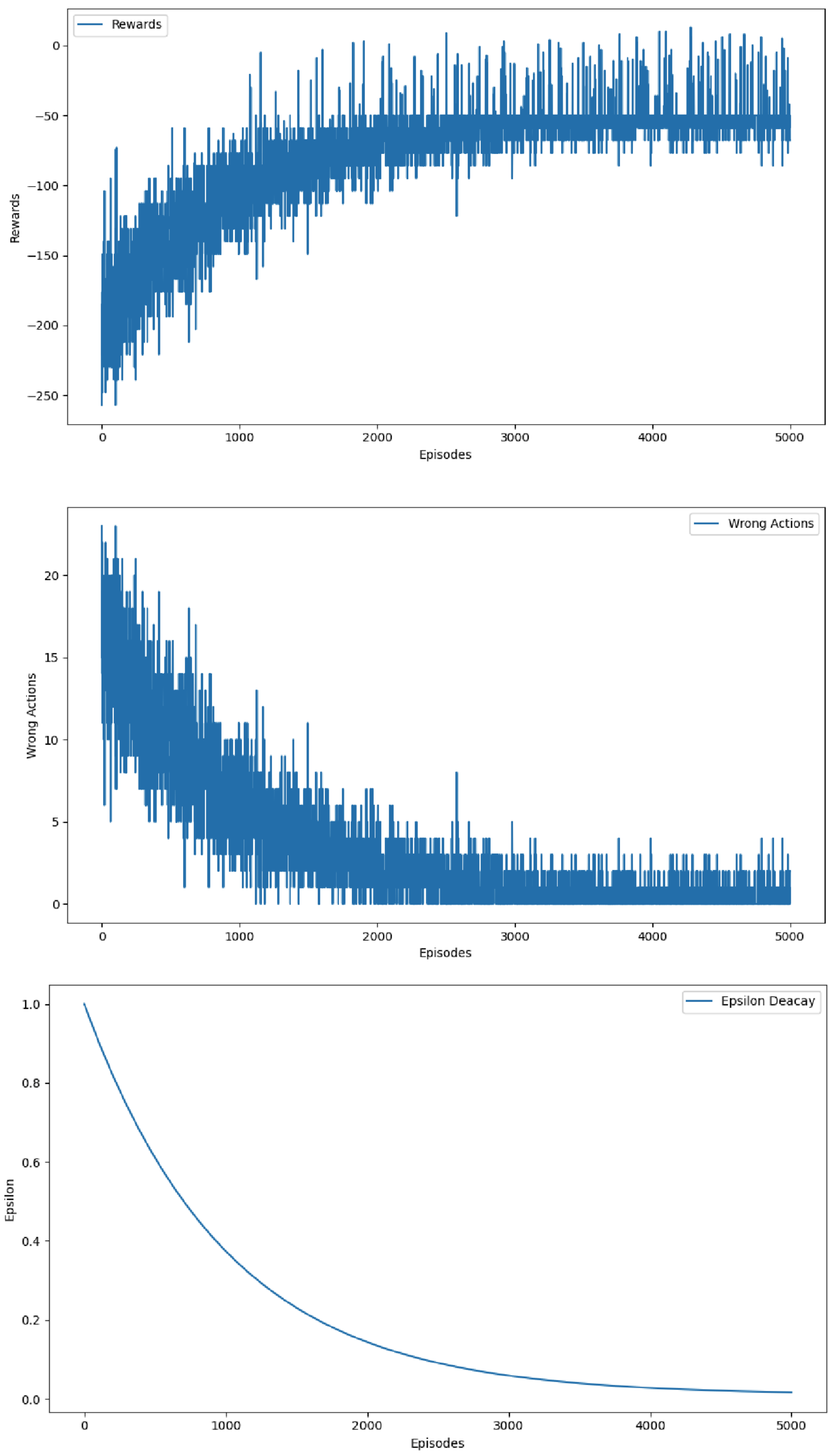
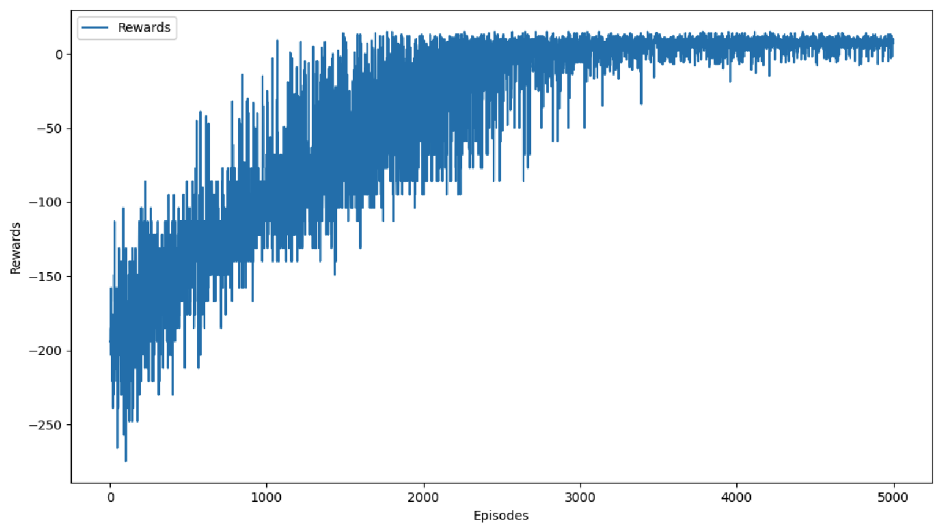
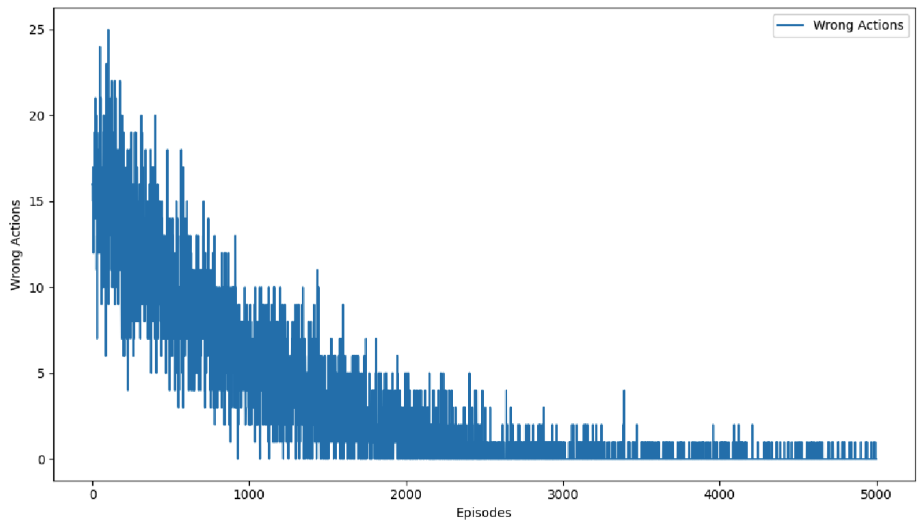


Figure 6: $\alpha = 0.01, \gamma = 0.99, decay\_rate = 0.001$





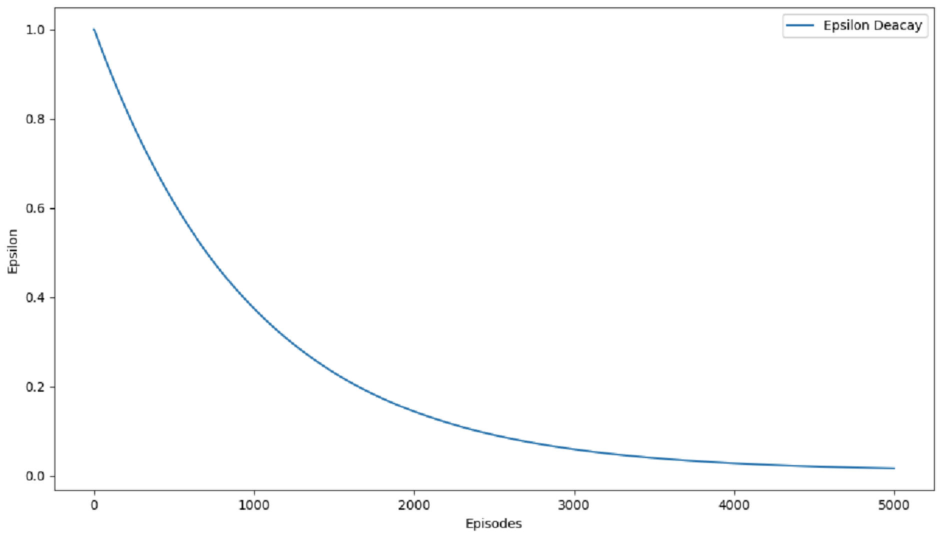
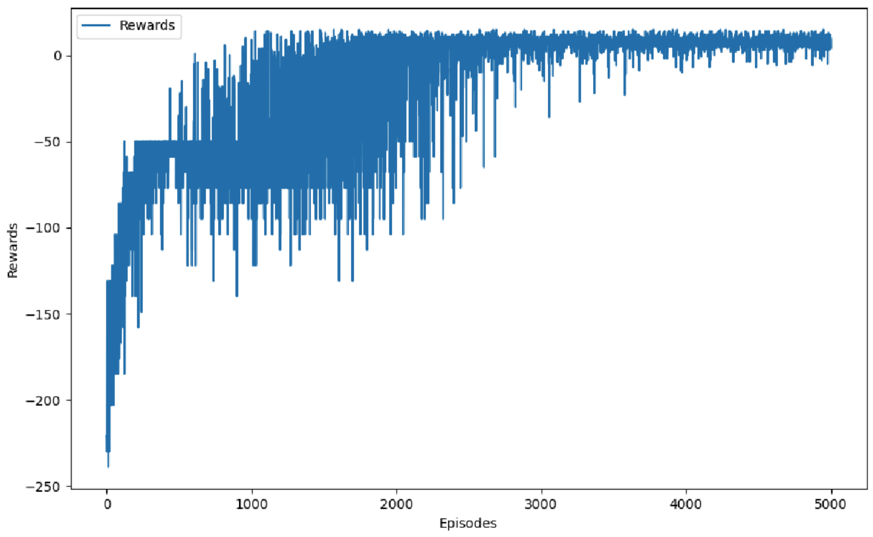
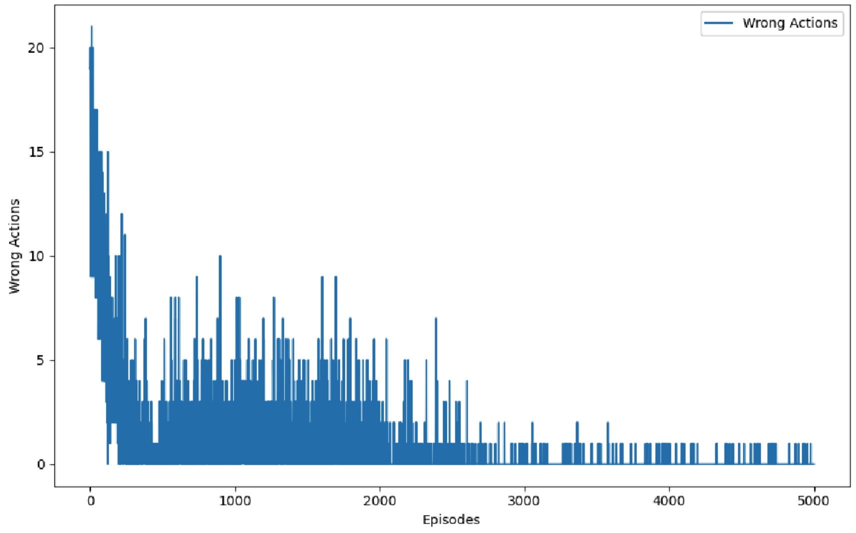


Figure 7: $\alpha = 0.1, \gamma = 0.99, decay\_rate = 0.001$





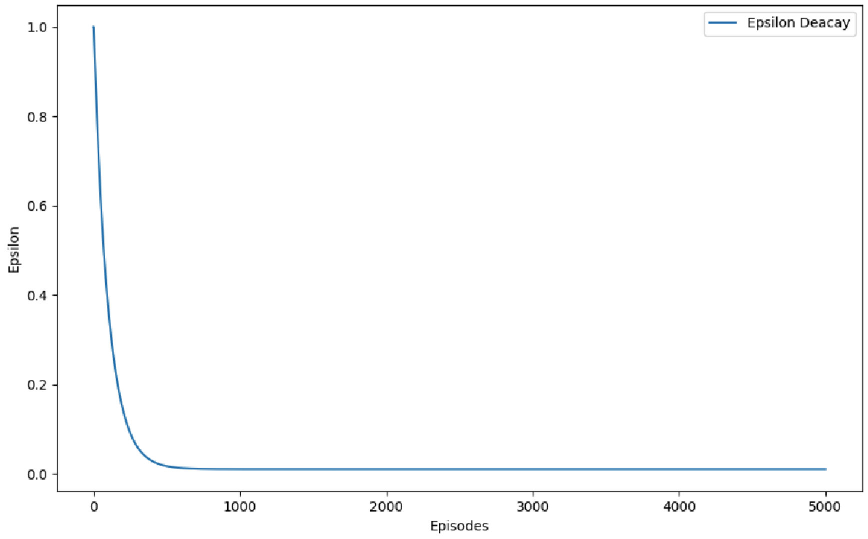


Figure 8: $\alpha = 0.1, \gamma = 0.99, decay\_rate = 0.01$

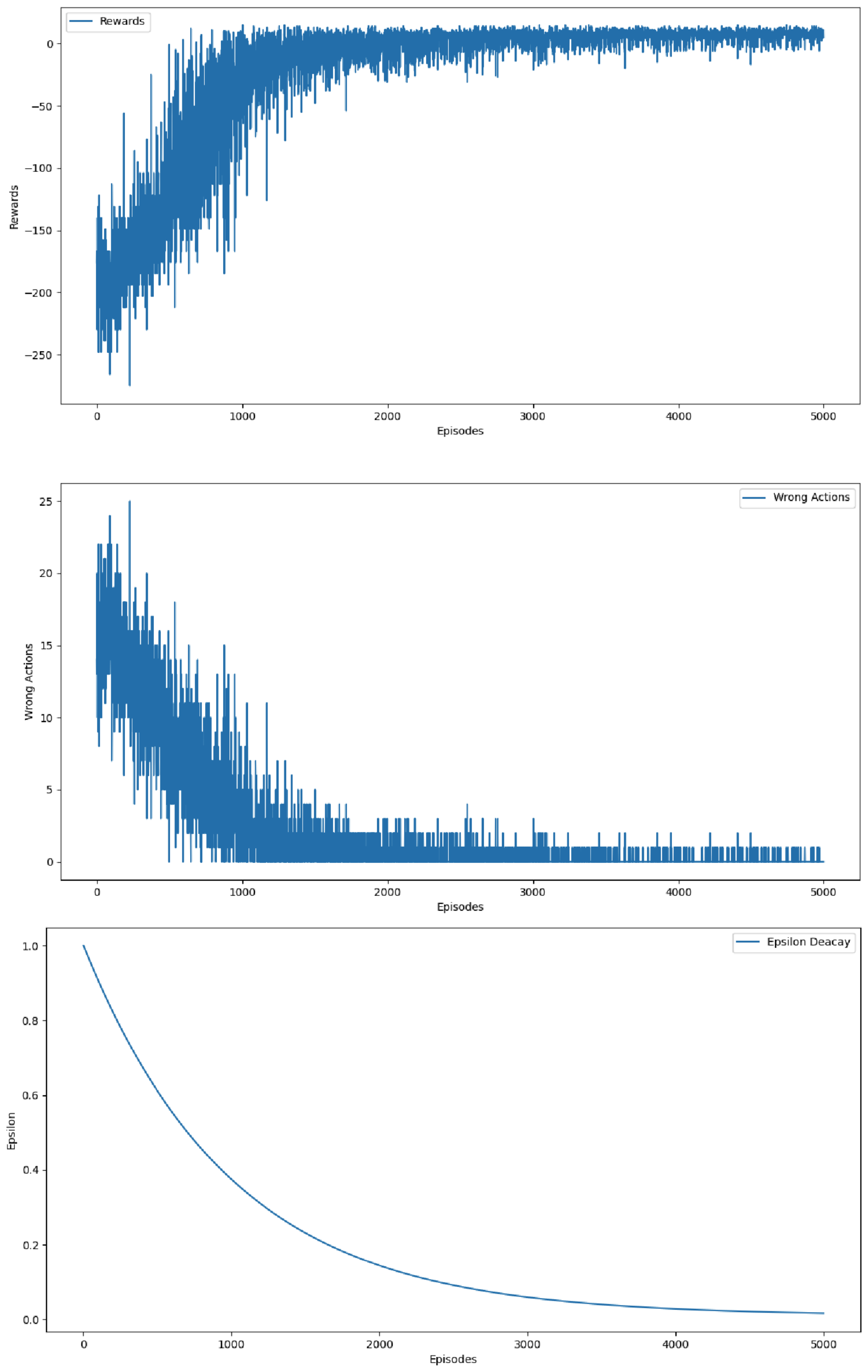


Figure 9: $\alpha = 0.5, \gamma = 0.99, decay\_rate = 0.001$

3.4 Q-function Approximation

As we said before, Q-function Approximation is an algorithm used to find the optimal action-selection policy by approximating the Q-function rather than explicitly storing it in a lookup table. Instead of maintaining a Q-table where the rows represent states and the columns represent actions, Q-function approximation algorithms employ a function approximator, such as a neural network, to map states to their corresponding Q-values. The parameters of the function approximator are learned through an iterative process using experiences collected from interacting with the environment. The Q-learning algorithm with Q-function approximation follows these steps:

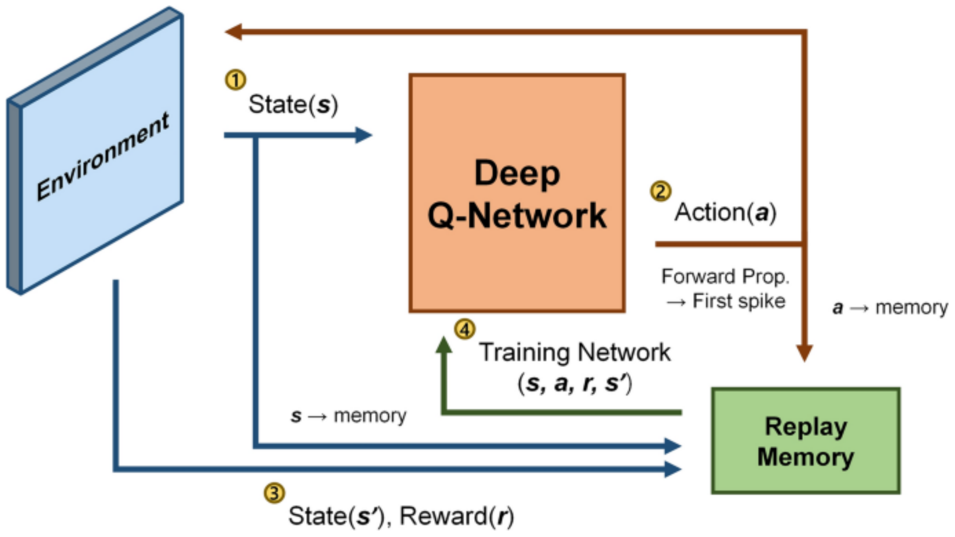


Figure 10: Q-function Approximation

In our solution, we created a function, namely the **Qlearn** function to do the learning . We first defined a set of constants:

```
EXP_MAX_SIZE=5000
BATCH_SIZE=80
GAMMA = 0.95
ALPHA = 0.001
EPS_MAX = 100
EPS_MIN = 1
decay_epsilon = 0.001
WRONG_ACTION = -10
cumulative_reward = 0
number_of_wrong_actions = 0
```

- **EXP_MAX_SIZE:** The maximum size of the experience replay buffer. It determines the number of experiences stored for the training.
- **BATCH_SIZE:** The batch size used for the training. It specifies the number of experiences sampled from the replay buffer at each training iteration.
- **GAMMA:** The discount factor used in the Q-learning algorithm. It determines the weight given to future rewards compared to immediate rewards.

- **ALPHA:** The learning rate for updating the Q-function approximation. It controls how much the Q-values are updated based on new experiences.
- **EPS_MAX:** The maximum exploration rate (epsilon) used in the epsilon-greedy exploration strategy. It represents the probability of selecting a random action during exploration.
- **EPS_MIN:** The minimum exploration rate (epsilon). As the algorithm progresses, the exploration rate decreases over time according to a decay strategy.
- **decay_epsilon:** The rate at which the exploration rate (epsilon) decreases over time. It determines how quickly the algorithm transitions from exploration to exploitation.
- **WRONG_ACTION:** The penalty or negative reward assigned when an incorrect action is taken. It helps discourage the agent from making suboptimal actions.
- **cumulative_reward:** The cumulative reward accumulated during an episode or a sequence of interactions with the environment.
- **number_of_wrong_actions:** The count of incorrect actions taken during an episode or training session.

In particular, we have builded this neural network:

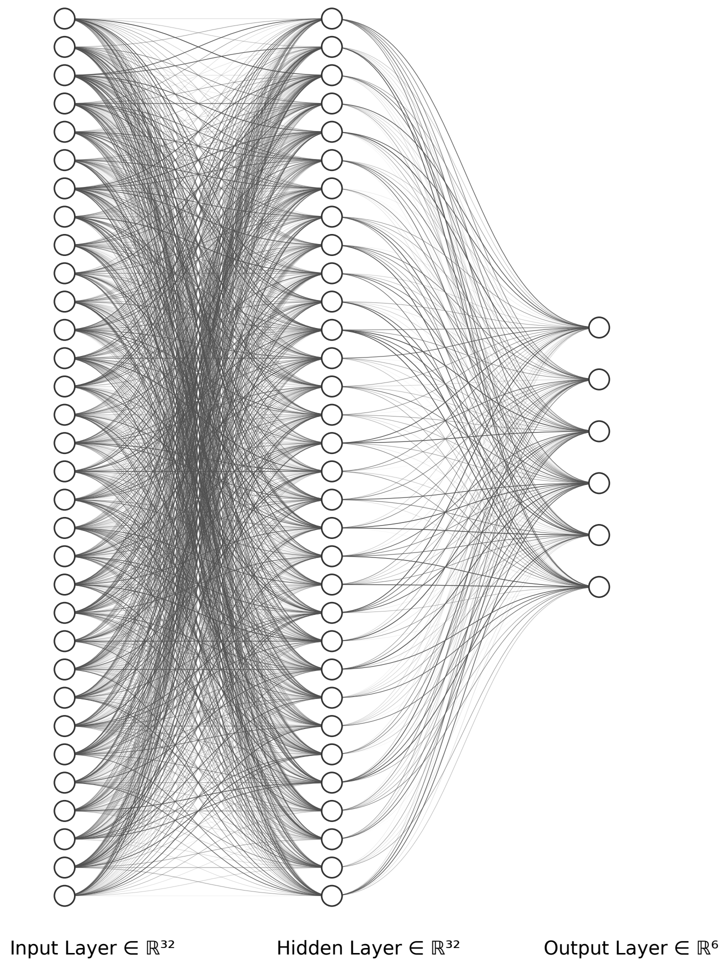

Input Layer $\in \mathbb{R}^{32}$
Hidden Layer $\in \mathbb{R}^{32}$
Output Layer $\in \mathbb{R}^{6}$

Figure 11: Neural Network

```
def create_model(enviroment):
    model = Sequential()
    model.add(Dense(32, activation='relu', input_shape=(1, )))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(enviroment.action_space.n, activation='linear'))
    model.compile(loss='mse', optimizer='adam')
    return model
```

The create_model function is responsible for creating a neural network model that approximates the Q-function for a given environment, taxi in our case. The function starts by initializing a sequential model, which represents a linear stack of layers. Each layer in the model is responsible for transforming the input data to produce meaningful outputs. In the case of Q-function approximation, we use dense layers, also known as fully connected layers. The dense layers are added to the model one by one. These layers consist of multiple neurons that are fully connected to the neurons in the previous layer. This connectivity enables the neural network to capture complex relationships between inputs and outputs. The activation functions used in the dense layers determine the non-linear behavior of the neurons and introduce non-linearity into the model. ReLU (Rectified Linear Unit) and linear activation functions are commonly used in Q-function approximation. Once all the dense layers are added, the model is compiled. Compilation involves specifying the loss function and optimizer for training the model. The loss function measures the discrepancy between the predicted Q-values and the target Q-values. In this case, Mean Squared Error (MSE) is used as the loss function. The optimizer is responsible for updating the model's weights during the training process to minimize the loss. The Adam optimizer, known for its efficiency and effectiveness, is commonly used in Q-learning algorithms.

```
def Qlearn(enviroment, model,number_of_episodes):
    #Experience Buffer
    experience = deque([],EXP_MAX_SIZE)

    #Parameters
    epsilon = EPS_MAX

    decay_epsilon = 0.001
    WRONG_ACTION = -10

    cumulative_reward = 0
    number_of_wrong_actions = 0
    episode = 1

    state, _ = enviroment.reset()

    while(episode <= number_of_episodes):

        selected_action = get_action(enviroment, epsilon, model, state)
        state_next, reward, terminated, truncated, _ = enviroment.step(selected_action)

        if reward == WRONG_ACTION:
            number_of_wrong_actions += 1
        cumulative_reward += reward

        # Record experience
        if len(experience)>=EXP_MAX_SIZE:
            experience.popleft()

        experience.append([state,selected_action,reward,state_next, terminated or truncated])

        state = state_next # update current state

        if terminated or truncated:
            if len(experience) >= BATCH_SIZE:
                train_model(experience,model)

            #debug information
            print("---------------------------------------episode ", episode)
            print("return=",cumulative_reward)
            print("epsilon=", epsilon)
            print("wrong actions=",number_of_wrong_actions)

            epsilon = EPS_MIN  + (EPS_MAX - EPS_MIN) * np.exp(-decay_epsilon*episode)
            episode += 1
            cumulative_reward = 0
            number_of_wrong_actions = 0
            state, _ = enviroment.reset()
    model.save("model_NN_taxi")
    enviroment.close()
```

In Qlearn we train the neural network, so we start by define an experience buffer that is used to store the agent's experiences during training. It keeps track of the agent's interactions with the environment, including the current state, selected action, received reward, next state, and termination information. The environment is initialized by obtaining the initial state through enviroment.reset(). This prepares the agent for interacting with the environment. The core of the algorithm resides in the Q-Learning Loop, which runs for a specified number of episodes. Within each episode, the agent selects an action based on the current state and the exploration-exploitation strategy determined by epsilon. The get_action function is responsible for this selection process. The selected action is executed in the environment using enviroment.step(selected_action). This provides the agent with the next state, the reward received, and information about termination or truncation. The agent records its experience, including the current state, selected action, obtained reward, next state, and termination information. These experiences are stored in the experience buffer, which retains a limited number of past experiences. The agent's current state is updated to the next state obtained from the environment. If the episode is terminated or truncated, certain actions are taken. The experience buffer is checked, and if it reaches its maximum capacity, the oldest experience is removed. If there are enough experiences in the buffer, the train_model function is called to update the Q-function approximation using a batch of experiences.

```
def get_action(enviroment, epsilon,model,state):
    #Exploration
    random_value = random.randint(1,100)
    selected_action = enviroment.action_space.sample()
    #Exploitation
    if random_value >= epsilon:
        predicted_actions = model.predict_on_batch(np.array([[state]]))[0]
        selected_action = np.argmax(predicted_actions)
    return selected_action
```

The get_action function balances exploration and exploitation. It randomly selects an action from the action space with a certain probability (epsilon) for exploration, but it exploits the learned Q-function approximation to choose the action with the highest predicted Q-value when the random value is below epsilon.

```
def train_model(experience,model):
    batch = random.sample(experience, BATCH_SIZE)
    dataset = np.array(batch)
    X = np.asarray(dataset[:,0])

    datasetY = []
    for current_state,current_action,current_reward,next_state, done in batch:

        predicted_actions = model.predict_on_batch(np.array([[current_state]]))[0]
        Q = predicted_actions[current_action]

        if done:
            predicted_actions[current_action] = current_reward
        else:
            predicted_next_actions = model.predict_on_batch(np.array([[next_state]]))[0]
            Qmax = predicted_next_actions[np.argmax(predicted_next_actions)]
            Q_new = Q + ALPHA*(current_reward + GAMMA * Qmax - Q)
            predicted_actions[current_action] = Q_new

        datasetY.append(predicted_actions)
    Y = np.asarray(datasetY)

    model.fit(X, Y, validation_split=0.2)
```

The train_model function is responsible for training the Q-function approximation model using a batch of experiences obtained from the experience buffer. The following steps are involved:

- **Batch Sampling:** A random batch of experiences is sampled from the experience buffer using random.sample(experience, BATCH_SIZE). The batch is converted to a NumPy array for efficient processing using np.array(batch).

- **Input Preparation:** The current states from the batch are extracted and assigned to X using np.asarray(dataset[:,0]). These states will serve as the input for the model.

- **Q-Value Update:** A list, datasetY, is created to store the updated Q-values for the corresponding actions in the batch. For each experience in the batch, the Q-value for the current action is obtained by predicting the Q-values for the current state using the Q-function approximation model:
  predicted_actions = model.predict_on_batch(np.array([[current_state]]))[0]. If the experience is a terminal state (done is True), the Q-value for the current action is updated directly to the current reward: predicted_actions[current_action] = current_reward. If the experience is not a terminal state, the Q-value is updated using the Q-learning update rule:

  – The Q-values for the next state are predicted using the model:
    predicted_next_actions = model.predict_on_batch(np.array([[next_state]]))[0].
  – The maximum Q-value among the predicted Q-values for the next state is obtained:
    Qmax = predicted_next_actions[np.argmax(predicted_next_actions)].
  – The updated Q-value for the current action is calculated using the Q-learning update formula:
    Q_new = Q + ALPHA*(current_reward + GAMMA * Qmax - Q).
  – The updated Q-value is assigned to the corresponding action in predicted_actions:
    predicted_actions[current_action] = Q_new.
  – The updated Q-values for the batch are appended to datasetY.

- **Output Preparation:** The list datasetY is converted to a NumPy array to serve as the target output for the model: Y = np.asarray(datasetY).

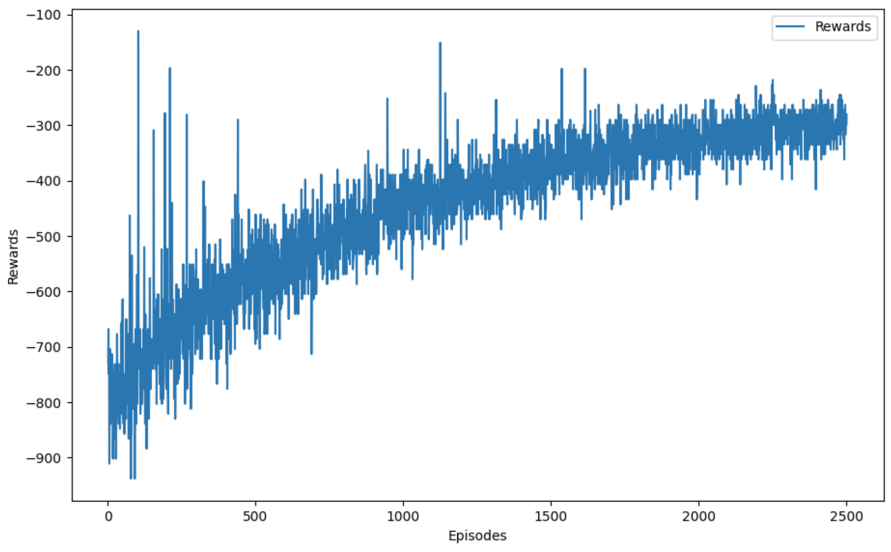
- **Model Training:** The model is trained using the prepared inputs (X) and target outputs (Y) from the batch using model.fit(X, Y, validation_split=0.2). The validation split refers to the process of splitting the available data into two separate sets: a training set and a validation set. The training process aims to minimize the discrepancy between the predicted Q-values and the target Q-values using the Mean Squared Error (MSE) loss function.

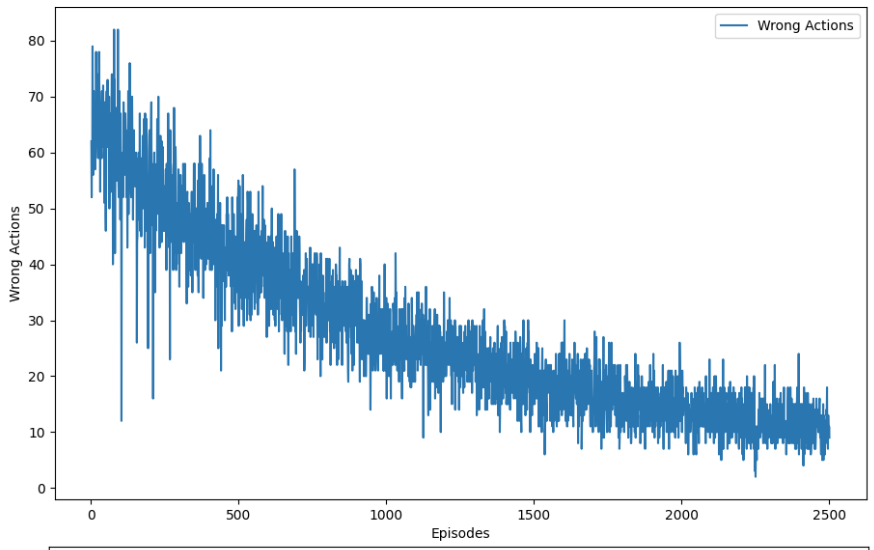
3.4.1 Q-function approximation performance

As for the Qtable, in order to improve the performance of the learning, the fine-tuning of the parameters used to calculate Q-function, such as **alpha**, **gamma** and **decay_rate** is necessary. The parameters are respectively the learning rate, the discount factor and the decay rate of the probability to exploit the model. We have tried different combination of these parameters in order to evaluate which is the best configuration. In particular, we tested the following configurations:

- $\alpha = 0.001, \gamma = 0.95, decay\_rate = 0.001$
- $\alpha = 0.001, \gamma = 0.99, decay\_rate = 0.001$
- $\alpha = 0.01, \gamma = 0.99, decay\_rate = 0.001$
- $\alpha = 0.01, \gamma = 0.5, decay\_rate = 0.001$

For each test, we plotted the cumulative reward, the number of wrong actions, and the epsilon decay. From the following charts we can see that the best parameter configuration is the first one, that is $\alpha = 0.001, \gamma = 0.95, decay\_rate = 0.001$.
This is because it takes **2500 episodes** to maximize the cumulative reward and to minimize the wrong actions, compared to all other configurations that achieved the result more slowly and less smoothly. Indeed, as we can see, in the first configuration we have fewer negative spikes in the charts (both for the cumulative reward and the wrong actions) than the other configurations.





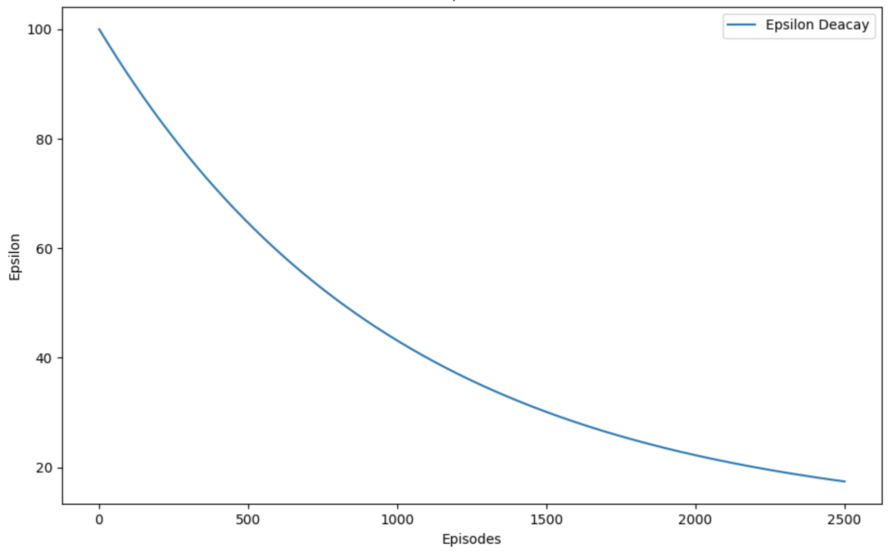


Figure 12: $\alpha = 0.001, \gamma = 0.95, decay\_rate = 0.001$

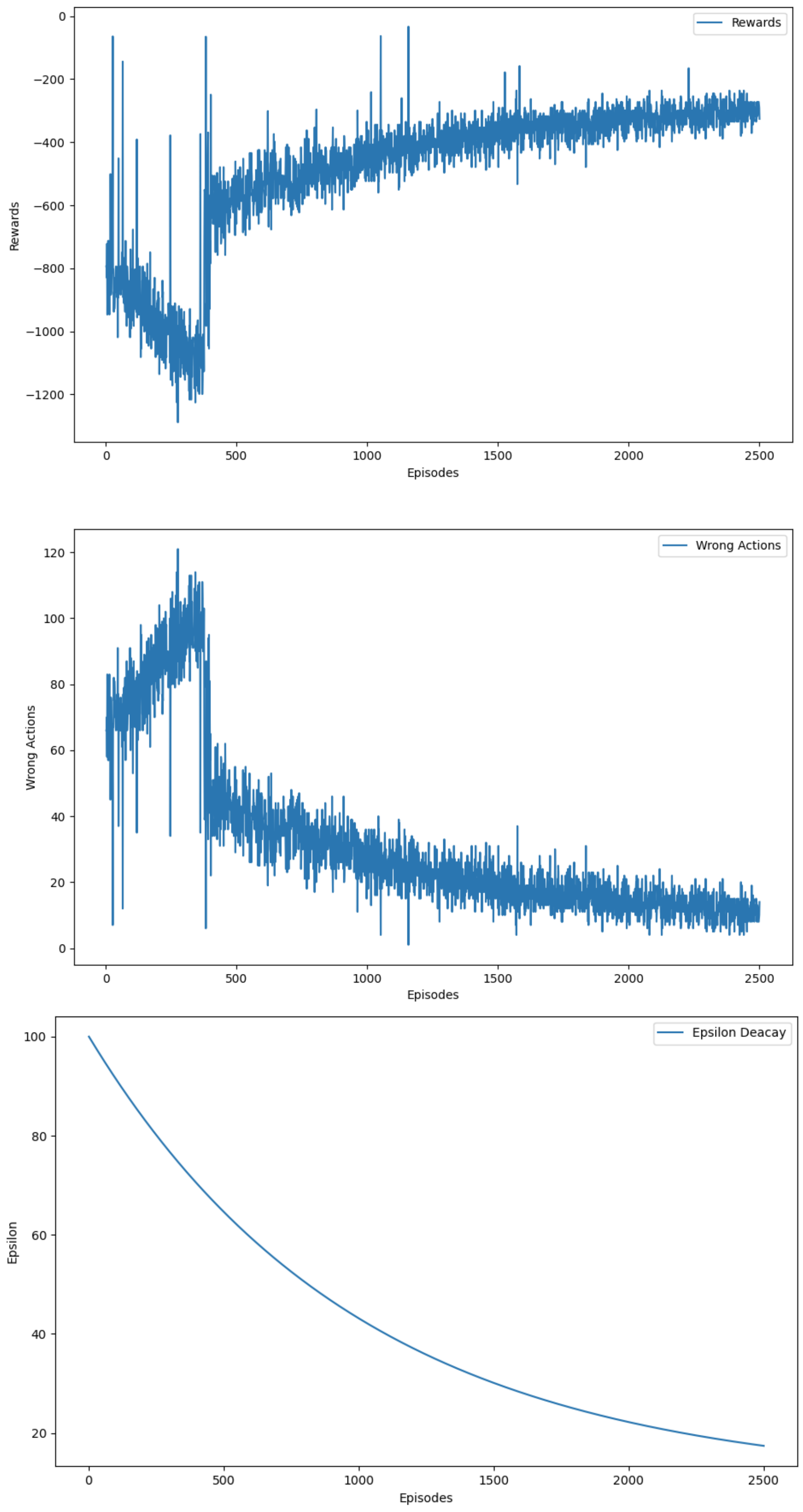
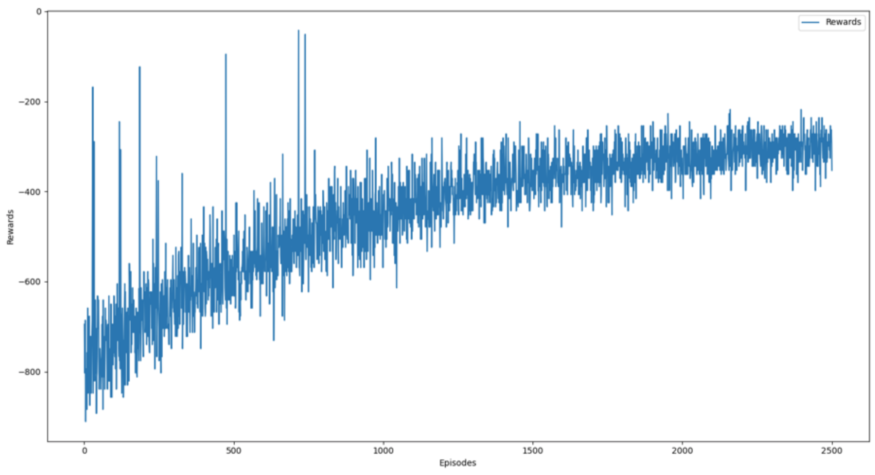
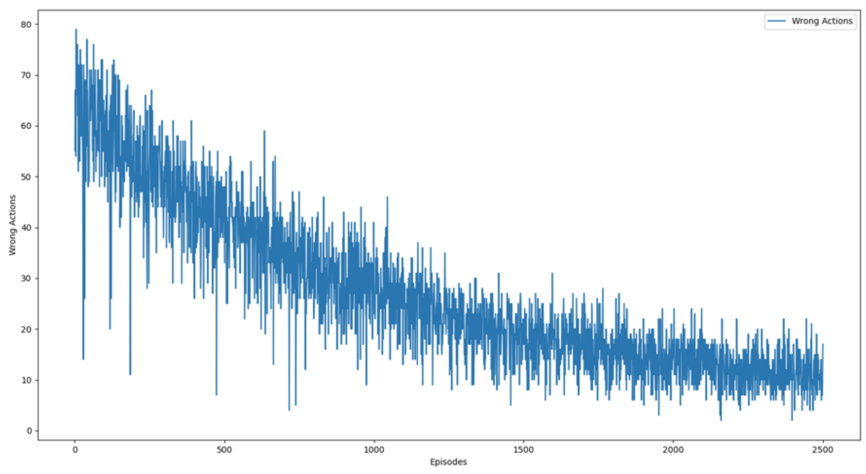


Figure 13: $\alpha = 0.001, \gamma = 0.99, decay\_rate = 0.001$





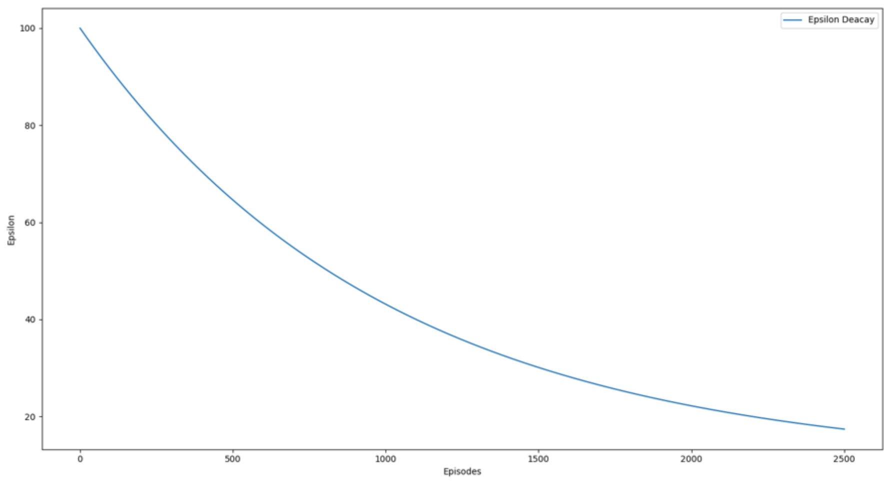


Figure 14: $\alpha = 0.01, \gamma = 0.99, decay\_rate = 0.001$

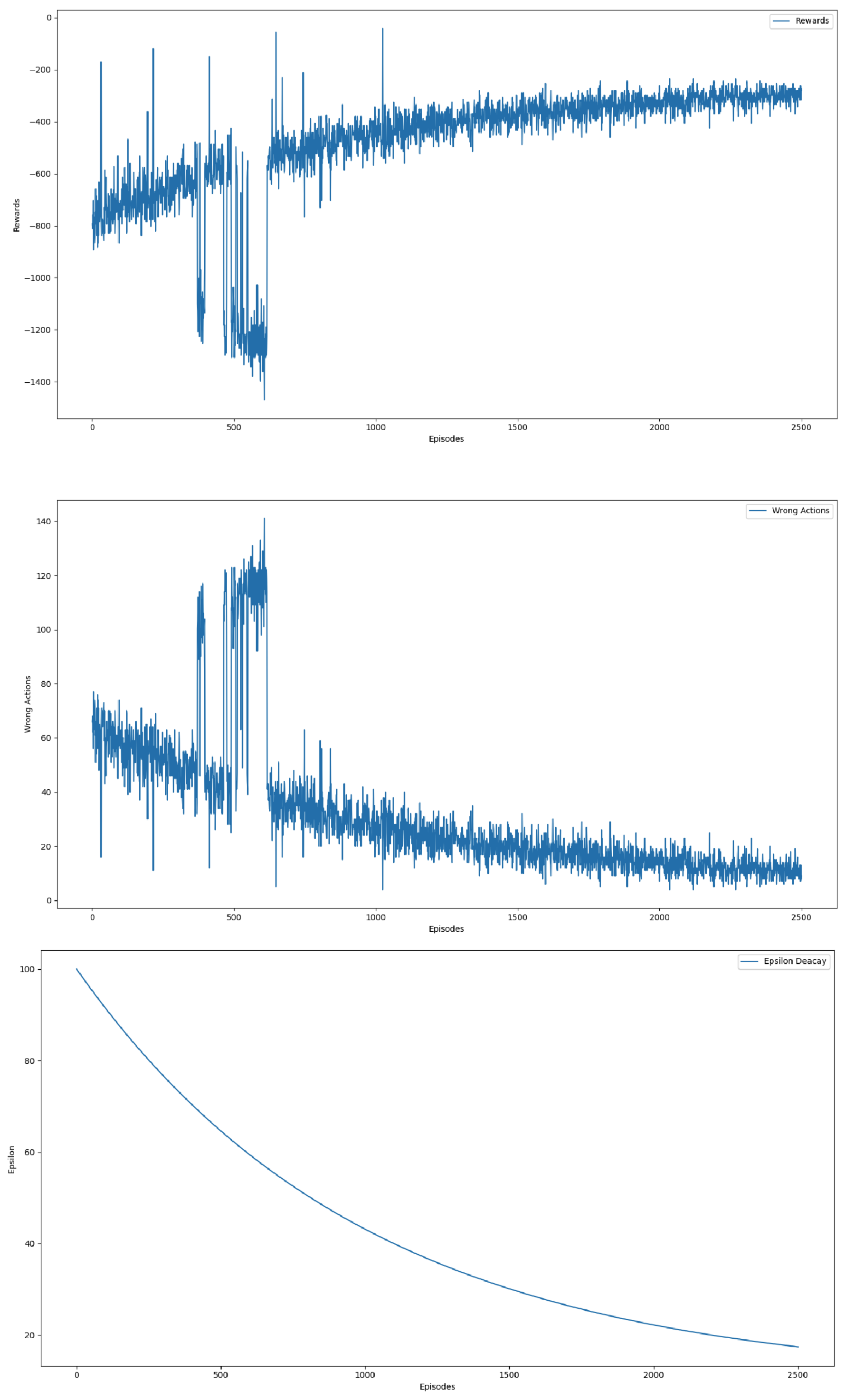


Figure 15: $\alpha = 0.01, \gamma = 0.95, decay\_rate = 0.001$

4 Conclusion

In conclusion, a comparison can be made between using a Q-table and a neural network to solve a reinforcement learning problem. We can use a Q-table when the state and action spaces are small and discrete, and the problem is relatively simple. On the other hand, we use a neural network when dealing with large or continuous state spaces, and when the problem is more complex.
Regarding the memory and computational requirements we have that Q-tables can consume significant memory, especially when the state and action spaces are large. As the number of states and actions increases, the storage requirements become impractical. Instead, the other approach typically require less memory as they use function approximation to estimate Q-values. However, they require computational resources for training, especially when dealing with complex networks and large state spaces.
Regarding the efficiency, Q-tables can achieve high efficiency since they explicitly store the Q-values for all state-action pairs. Each update directly affects the value of a specific state-action pair, enabling faster convergence. Q-function approximation typically require more samples to learn effectively. However, techniques like experience replay can improve sample efficiency.
Finally, the Q-function approximation approach can be more challenging to train and requires careful tuning of hyper-parameters and network architecture. It may suffer from instability during training, including issues like overestimation or underestimation of Q-values and sensitivity to initial conditions.