SAPIENZA
UNIVERSITÀ DI ROMA

A Tabular Q-learning and a Deep Q-Network approach for solving Taxi

Machine Learning Project

Faculty of Information Engineering, Computer Science and Statistics

M.Sc. Engineering in Computer Science

Academic Year 2023/2024

Presented by:
Gloria Marinelli, 2054014
Alessandro Pisent, 2085678

Contents

1 Introduction

Reinforcement learning (RL) is a method for addressing problems that involve continuous decision-making.

It allows an agent to discover an optimal behavioural policy through interaction with the environment, learning which actions lead to the highest cumulative rewards.

The agent interacts with the environment and performs multiple actions to determine the optimal policy by receiving reward feedback for each action.

At the heart of reinforcement learning is the concept of a **policy**. It is a strategy that dictates the agent's actions in any given state. The agent refines this policy over time by receiving feedback from the environment in the form of rewards. Reinforcement learning algorithms help the agent to balance **exploration** (trying new actions to discover their outcomes) and **exploitation** (choosing actions known to yield the highest rewards) in order to find the **optimal policy**.

Two common approaches to reinforcement learning are the **Q-table method** and the **Q-function approximation**. In the context of this project, we will implement a Reinforcement Learning Agent based on Q-learning, specifically focusing on two approaches: **Tabular Q-learning** and **Deep Q-Network (DQN)**, in its simplest form, which utilises a single neural network to approximate the Q-function.

1.1 Tabular Q-learning

The Q-table method is the simplest form of Q-learning. It uses a table (**Q-table**) to store Q-values for every possible state-action pair in the environment. The **Q-value** represents the expected cumulative reward the agent can achieve by taking a particular action in a given state and then following the optimal policy thereafter.

Initially, the Q-table is empty, and the agent explores the environment to populate it. The agent selects actions based on an **exploration-exploitation strategy**, such as **epsilon-greedy**, where it occasionally chooses a random action to explore the environment and at other times selects the action with the highest known Q-value.

After taking an action and observing the reward, the agent updates the corresponding Q-value using the **Bellman equation**, which adjusts the Q-value based on the immediate reward and the highest Q-value of the subsequent state.

This iterative process continues until the Q-values **converge**, meaning the agent has learned the optimal policy. While this method is effective for small, discrete environments, it becomes impractical for large or continuous environments because it requires storing and updating Q-values for every possible state-action pair, which can be computationally expensive and memory-intensive.

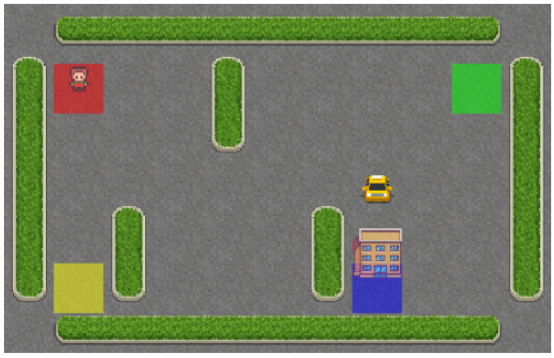
1.2 Q-Function Approximation (Deep Q-Networks)

As environments grow in complexity, Q-function approximation uses a function, often a parametric model such as a **neural network**, to estimate the Q-values. This approach allows the agent to generalise across large or continuous state-action spaces by learning patterns from the data, rather than storing individual Q-values for each state-action pair.

One popular approach is the **Deep Q-Network (DQN)**, where a neural network is used to approximate the Q-function. In this setup, the agent still employs an exploration-exploitation strategy, similar to tabular Q-learning. However, instead of updating a table, the agent trains the neural network using a **loss function** that incorporates the Bellman equation. The network adjusts its weights to minimise the difference between predicted Q-values and the actual observed rewards.

Effective training of a **DQN** often requires techniques like **experience replay** (storing past experiences and sampling them randomly during training) and a **neural network** used to approximate the Q-table.

2 Taxi-Environment



The **Taxi environment**, which forms part of Gymnasium, is a toy text environment that is frequently employed to illustrate reinforcement learning algorithms. It simulates a simplified scenario that involves navigating to passengers in a grid world, picking them up and dropping them off at one of four locations.

States

The environment consists of **500 discrete states**:

- **25 taxi positions**: The taxi can be located at any square in a 5x5 grid.
- **5 possible passenger locations**: Red, Green, Yellow, Blue, and "In taxi."
- **4 destination locations**: Red, Green, Yellow, Blue.

The state is encoded as an integer, calculated as:

```
((taxi_row * 5 + taxi_col) * 5 + passenger_location) * 4 + destination
```

There are **404 states** reachable within an episode: **400 valid states** during the task and **4 additional states** appear when the passenger and the taxi are both at the destination at the end of the episode.

Actions

The action space consists of **6 discrete actions**, represented by integers {0, 5}:

- **Move South (0)**: Move the taxi one square down.
- **Move North (1)**: Move the taxi one square up.
- **Move East (2)**: Move the taxi one square right.
- **Move West (3)**: Move the taxi one square left.
- **Pickup (4)**: Pick up the passenger (if at the same location).
- **Drop off (5)**: Drop off the passenger (if at the correct destination).

Rewards

- **-1** for every step taken (unless a different reward is triggered).

- **+20** for successfully dropping off the passenger at the correct destination.
- **-10** for illegal attempts to pick up or drop off the passenger.

Settings

- **Grid World**: A 5x5 grid with four designated locations Red, Green, Yellow, and Blue.
- **Starting State**:
  - Taxi starts at a random position.
  - Passenger starts at one of the designated locations.
- **Observation Space**: Encoded as a discrete state based on taxi position, passenger location, and destination.
- **Action Mask**: Specifies whether an action will change the state, helping to avoid invalid or "noop" actions (e.g., moving into a wall).

Goals

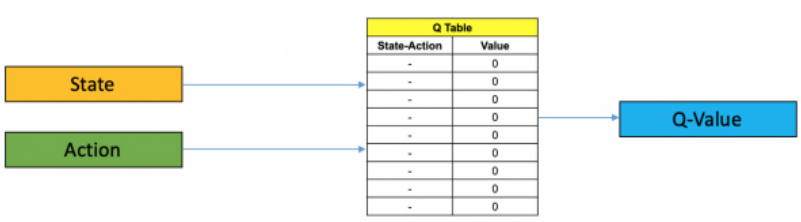
- **Primary Goal**:
  - Move the taxi to the passenger's location.
  - Pick up the passenger.
  - Transport the passenger to the designated drop-off location.
  - Drop off the passenger at the correct destination.
- **End of Episode**:
  - The episode terminates upon successful drop-off.
  - The episode can also be truncated if it exceeds a length of **200 steps** (with a time limit wrapper).

3 Solution Adopted

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

3.1 Q-Table

As previously mentioned, Q-learning is an algorithm used to find the optimal action-selection policy using a Q-function. The Q-table is a lookup table where rows represent states, and columns represent actions. The Q-table helps us choose the best action for each state to maximize the expected cumulative reward.



3.2 Tabular Q-learning

We implemented a Q-learning algorithm, where a single Q-table was used to approximate the optimal Q-values for a given environment. The algorithm is off-policy, meaning it learns the optimal policy independently of the current policy followed during training.

The **update rule** for the Q-table is:

$$Q(S,A) = Q(S,A) + \alpha \left(R + \gamma \max_{A'} Q(S',A') - Q(S,A) \right) \tag{1}$$

where:

- S is the current state, and A is the action taken
- R is the reward received after taking action A
- S' is the next state after performing action A
- A' represents the next action, which will give the max value to the future reward
- α is the learning rate ($\alpha \in (0,1)$)
- γ is the discount factor ($\gamma \in (0,1)$), which controls the balance between short-term and long-term rewards

The agent explores the environment and updates the Q-values iteratively, eventually converging toward an optimal policy.

3.2.1 Epsilon-Greedy Policy

The epsilon-greedy policy is a simple **action-selection strategy**. The agent chooses the action with the highest Q-value for the current state most of the time (**exploitation**), but with probability ε, it selects a random action (**exploration**). This encourages exploration of the environment, balancing between exploration and exploitation.

The algorithm proceeds with the following steps:

- Initialize the Q-table, learning rate α, discount factor γ, and a function to decrease ε: $f(\text{current episode})$
- For each episode:
  - Initialize the starting state S
  - Repeat until the terminal state is reached:
    - Observe the current state S
    - Choose an action A based on the **ε-greedy policy**, where $\varepsilon = f(\text{current episode})$
    - Execute action A and observe the next state S' and reward R
    - Update the Q-value for the current state-action pair (S,A) using the update rule (1)
    - Set state $S = S'$
- End the episode when the terminal state is reached

After experimenting with different **hyperparameters**, we found that the best configuration for our problem was:

```
alpha = 0.1      # Learning rate
gamma = 0.80     # Discount factor

num_episodes = 10_000
max_steps = 1000   # Max steps per episode
Q = training(num_episodes, select_action_greedy, steady_decrease_epsilon, alpha, gamma,
"Epsilon with Greedy policy", max_steps, env)
```

Additionally, for decreasing ε, a linear decrease over time provided better results compared to exponential or logarithmic decreases.

```
def select_action_greedy(Q, state, episode, max_episodes, update_epsilon, env):
    """
    Selects an action using the epsilon-greedy policy.

    Parameters:
    - Q: The Q-table.
    - state: The current state.
    - episode: The current episode number (0-indexed).
    - max_episodes: The total number of episodes.
    - update_epsilon: A function that updates the epsilon value.

    Returns:
    - action: The selected action.
    """

    # the actual function to update epsilon in our case is :
    # update_epsilon(): 1.0 - episode / max_episodes
    epsilon = update_epsilon(episode, max_episodes)

    if torch.rand(1).item() < epsilon:  # Exploration
        return env.action_space.sample()
    else:  # Exploitation
        return torch.argmax(Q[state]).item()
```

3.2.2 Softmax Policy

The softmax policy offers a **more balanced exploration strategy**. Instead of selecting the action with the highest Q-value deterministically, it assigns **probabilities to actions** based on their Q-values. Higher Q-values are given higher probabilities, but the agent still has a chance to explore other actions. This promotes both **exploration** and **exploitation**.

In addition, the "temperature" parameter b in the softmax function is adjusted over time. Initially, b is large to encourage **exploration**, but it decreases exponentially as training

progresses, making the agent's actions more deterministic, focusing on exploiting learned Q-values in later episodes.

The softmax algorithm proceeds with the following steps:

- Initialize the Q-table, learning rate α, discount factor γ, and a function to decrease b: $f(\text{current episode})$
- For each episode:
  - Initialize the starting state S
  - Repeat until the terminal state is reached:
    - Observe the current state S
    - Choose an action A based on the **softmax policy**, where $b = f(\text{current episode})$
    - Execute action A and observe the next state S' and reward R
    - Update the Q-value for the current state-action pair (S,A) using the update rule (1)
    - Set state $S = S'$
- End the episode when the terminal state is reached

Through **hyperparameter** tuning, we found the optimal configuration for our problem:

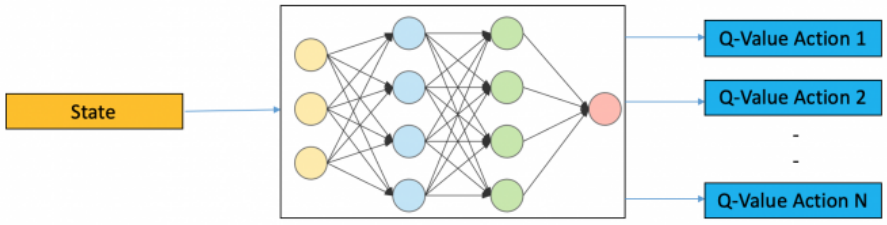
```
alpha = 0.1      # Learning rate
gamma = 0.80     # Discount factor

num_episodes = 10_000
max_steps = 1000   # Max steps per episode
Q = training(num_episodes, select_action_softmax, dynamic_base, alpha, gamma,
"Base of softmax (Softmax policy)", max_steps, env)
```

Additionally, for decreasing b, we used an exponential decrease during training.

```
    def select_action_softmax(Q, state, episode, max_episodes, update_base, env):
    """
    Selects an action using the softmax policy.

    Parameters:
    - Q: The Q-table.
    - state: The current state.
    - episode: The current episode number (0-indexed).
    - max_episodes: The total number of episodes.
    - update_base: A function that updates the base value.

    Returns:
    - action: The selected action.

    """

    #probs = torch.nn.functional.softmax(Q[state], dim=0)

    #  Calculate the decayed base value
    # update_base(episode, max_episodes) : max ( 20 * (0.20 ** (episode / max_episodes)),  math.e)


    base = update_base(episode, max_episodes)


    pow = torch.pow(base, Q[state])
    probs = pow / torch.sum(pow)

    return torch.multinomial(probs, 1).item()
```

3.3 Deep Q-learning

Deep Q-learning (DQN) is an extension of Q-learning that employs **neural networks** to approximate the Q-function. Instead of storing Q-values in a table (which is infeasible for large state spaces), a **deep neural network** is used to estimate the action-value function.



In line with the project specifications, our implementation employed a single neural network.

3.3.1 Algorithm

- Initialize replay buffer R
- Initialize Q-network $Q(S,A;\theta)$ with random weights θ
- Set parameters:
  - Learning rate α
  - Discount factor γ
  - Exploration rate ε
  - Number of episodes
- For each episode:
  - Initialize starting state S
  - Repeat until the terminal state is reached:
    - Choose action A based on the ε-greedy policy with respect to $Q(s,a;\theta)$
    - Execute action A, observe next state S' and reward R
    - Store the experience (S,A,R,S') in replay buffer R
    - Set state $S = S'$
  - If the number of elements in the replay buffer is greater than or equal to the mini-batch size:
    - Sample a mini-batch of experiences (S_j,A_j,R_j,S'_j) from replay buffer R
    - For each experience in the mini-batch:
      - Compute the target Y_j:

$$Y_j = R_j + \gamma \max_{A'} Q(S'_j,A';\theta)$$

    - Perform a gradient descent step on the **MSE loss function** $(Y_j, Q(S_j,A;\theta))$, using **ADAM** as the **optimizer**

- End the episode when the terminal state is reached

The Deep Q-Networks follow the **Bellman Equation**:

$$Q(S,A) = R + \gamma \max_{A'} Q(S',A';\theta) \tag{2}$$

where:

- S represents the current state of the environment
- A represents the current selected action
- R represents the reward obtained by performing the action A in the current state S of the environment
- S' represents the next state of the environment, obtained after having performed the current selected action A
- A' represents the next action, which will give the max value to the future reward
- γ represents the discount factor
- θ represents the wights of the neural network

So given the Bellman Equation, the **loss function** is the **MSE** and can be written as:

$$L(\theta) = (R + \gamma \max_{A'} Q(S',A';\theta) - Q(S,A;\theta))^2 \tag{3}$$

Using Gradient descent, and Adam optimizer we trained this model.

4 Experimental Results

Here the plots with the results of the training of our models. Our code also makes a video of the agent playing the game after it has finished learning as a visual test.

4.1 Tabular Q-learning

4.1.1 Epsilon-Greedy Policy

Here are the results of the training using the Epsilon-Greedy Policy and Tabular Q-learning.

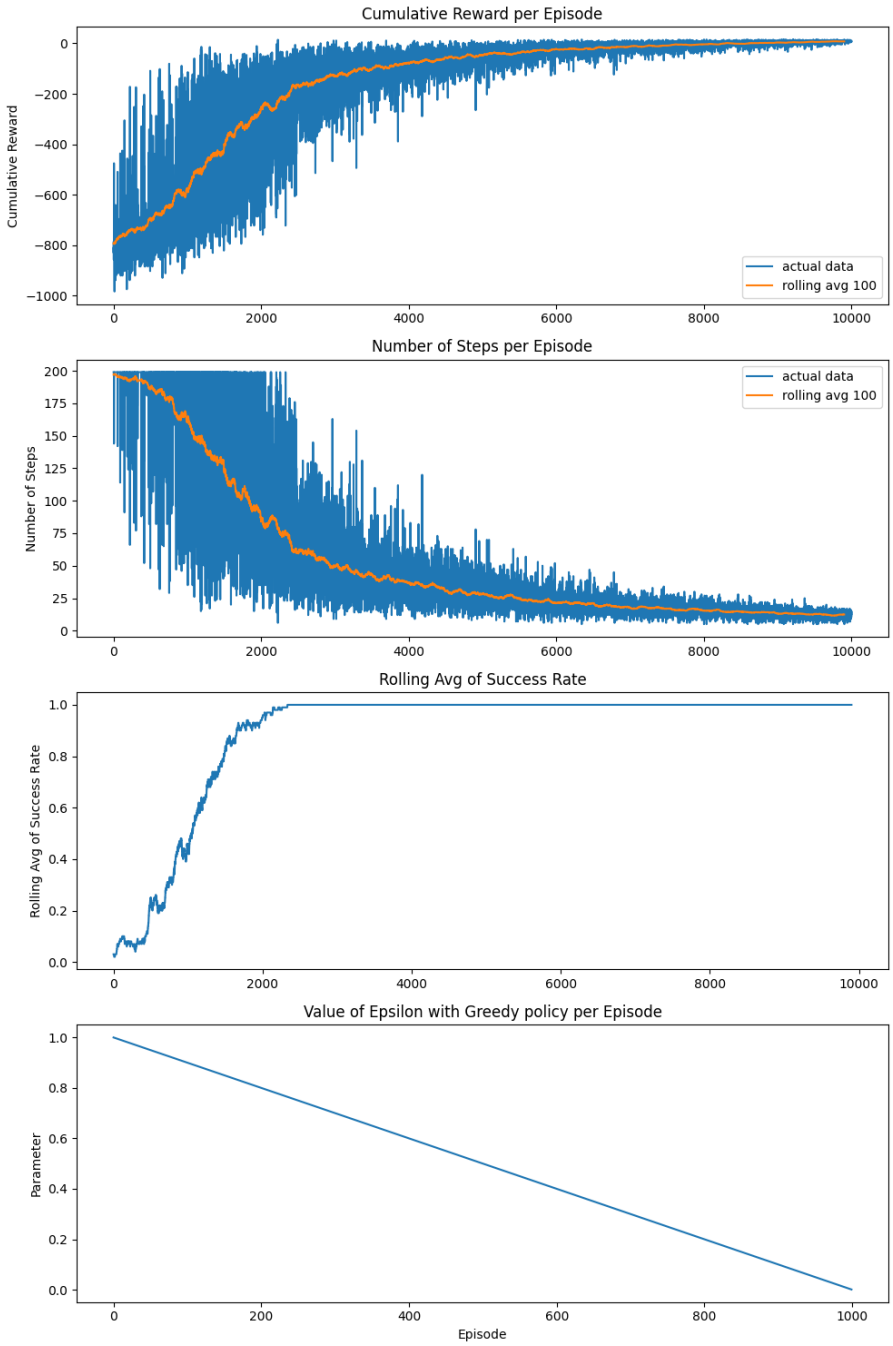


Figure 1: the Greedy policy

4.1.2 Softmax Policy

Here there are the result of the training using the Softmax Policy and Tabular Q-learning.

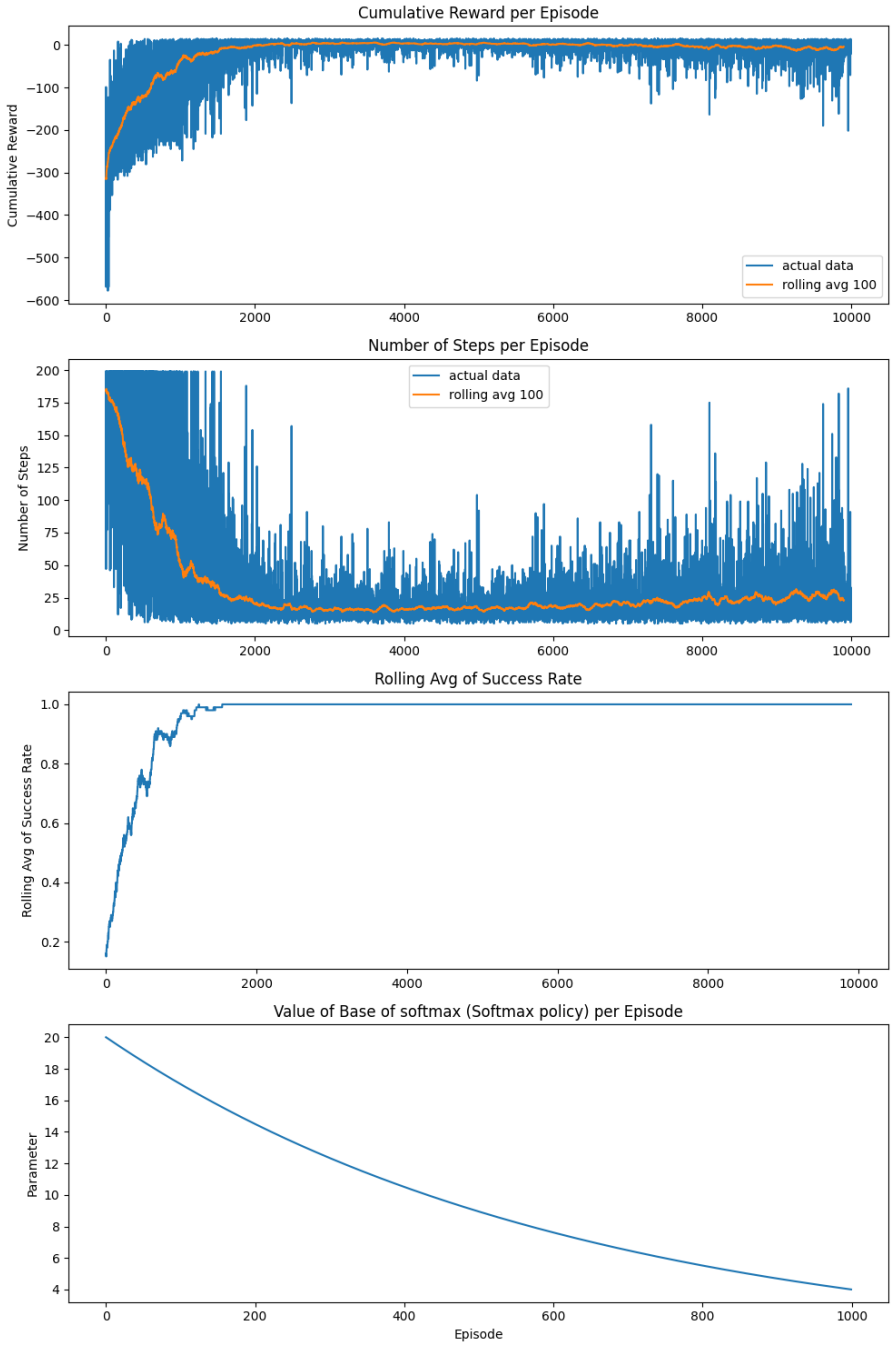


Figure 2: the Softmax policy

4.2 Deep Q-learning

Here is the graph for the result of the training of Deep Q-learning. Regarding the loss function plot, we also decided to plot the cumulative loss over the last 100 episodes to visualise the descending better.

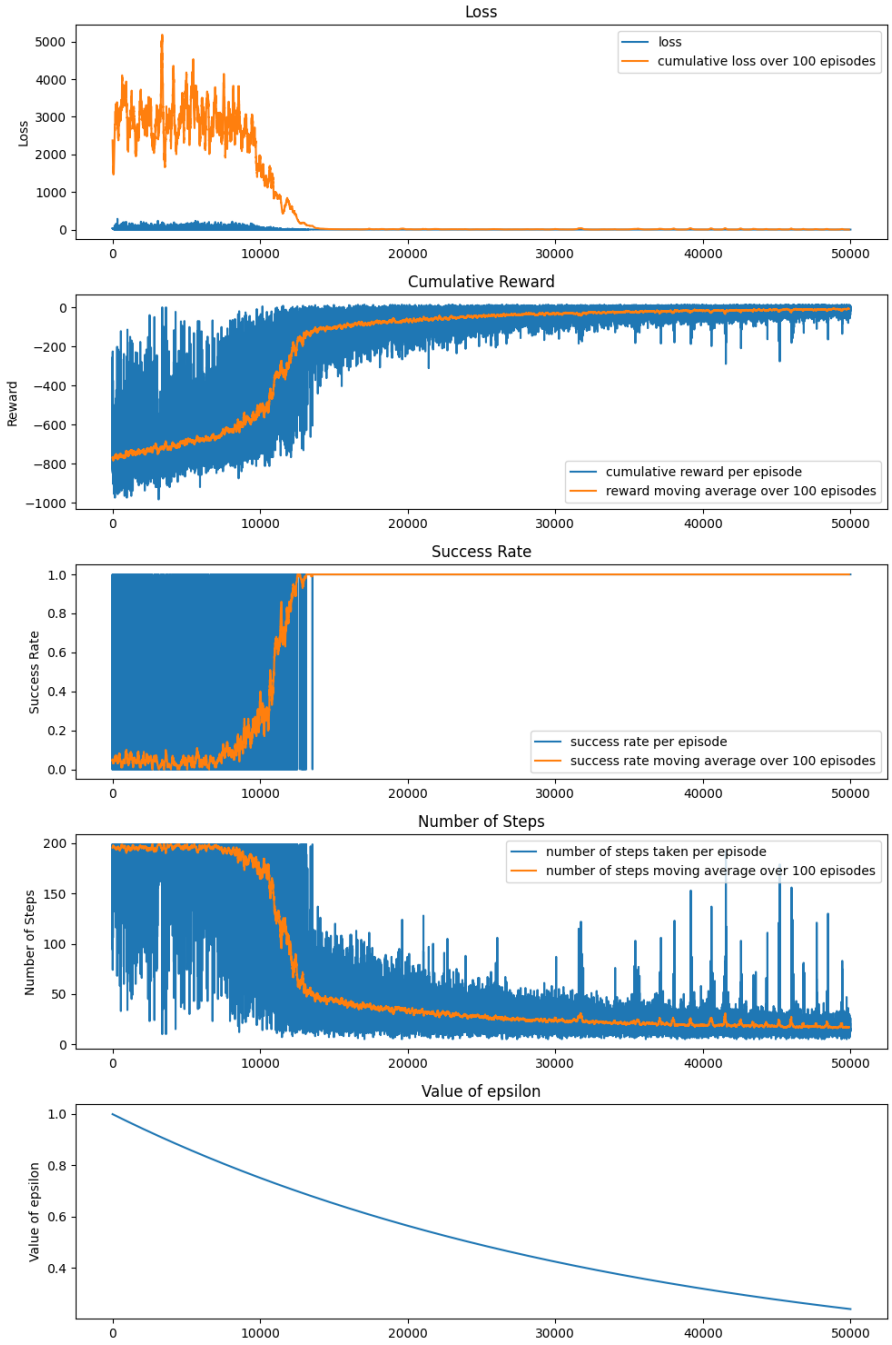


Figure 3: the DQN