Reinforcement Learning on a Gymnasium environment

Machine Learning project

Presented by: Giovanni Buracci
Submitted to: Prof. Fabio Patrizi

Academic Year 2023/2024

Contents

1. Introduction

In this report, we describe how it is possible to solve the Taxi-v3 environment using two popular reinforcement learning (RL) approaches: **Tabular Q-Learning** and **Deep Q-Learning (DQL)**.

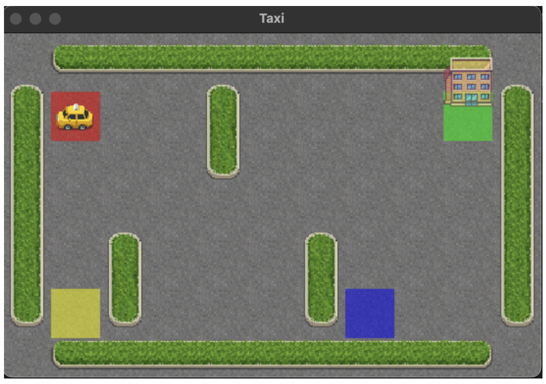
The goal of the project is to compare these methods in terms of performance, design, and effectiveness in solving the Taxi-v3 environment.

This document outlines the problem, the methods used, the architectures of the algorithms, and experimental results obtained from the learning process.

---

2. Problem Statement

The **Taxi-v3** environment is a classic RL problem where an agent (taxi) needs to navigate a grid world, pick up passengers from predefined locations, and drop them off at their destinations.



The agent interacts with the environment through actions such as moving in different directions, picking up, and dropping off passengers. The agent receives rewards based on its actions and whether they are taking it closer to or further from solving the problem.

2.1 Taxi-v3 Environment

- **State space**: The state space consists of 500 unique states, each representing the taxi's position, the passenger’s position, and the destination location.
- **Action space**: There are six possible actions: move up, down, left, right, pick up, and drop off.
- **Rewards**: The agent receives a reward of -1 for each time step, a reward of -10 for illegal pickup/dropoff actions, and +20 when it successfully delivers a passenger.

The challenge is for the agent to find an optimal policy that minimizes the number of steps and actions needed to pick up and drop off passengers.

---

3. Tabular Q-Learning

3.1 Overview

Tabular Q-Learning is a simple RL algorithm where we store and update a Q-table that contains the expected future rewards for each state-action pair. This approach is based on the Bellman Equation, where the Q-values are updated iteratively after each interaction with the environment.

3.2 Q-Learning Algorithm

The update rule for Q-Learning is:

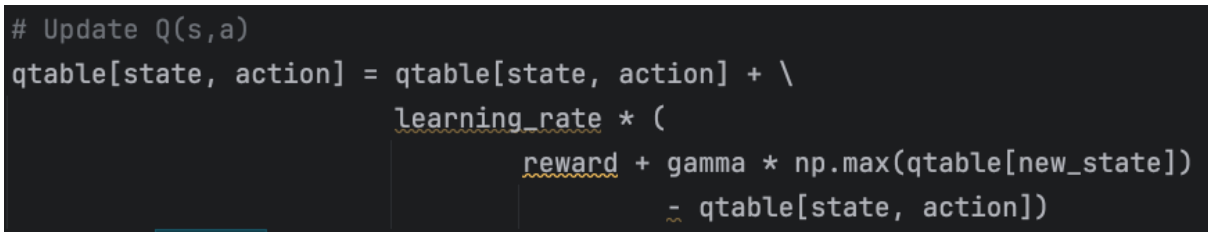
$$Q_{new}(s_t, a_t) = Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

Where:

- st is the current state, and at is the action taken.
- rt is the reward obtained.
- s' is the new state after taking action aaa.
- α is the learning rate, and γ is the discount factor.

The learning rate controls how fast the agent tends to learn a specific policy, regardless how optimal it is.
The discount factor, instead, manages how much the agent takes into consideration future rewards rather than immediate reward.

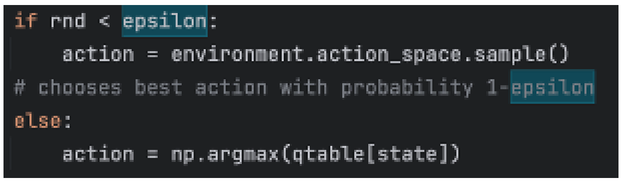
The algorithm updates the Q-table based on the reward and the highest expected reward for the next state.



```
# Update Q(s,a)
qtable[state, action] = qtable[state, action] + \
                        learning_rate * (
                            reward + gamma * np.max(qtable[new_state])
                                - qtable[state, action])
```

3.3 Choosing actions

At each iteration, the action is selected using an ε-greedy policy that manages exploration vs exploitation balance.



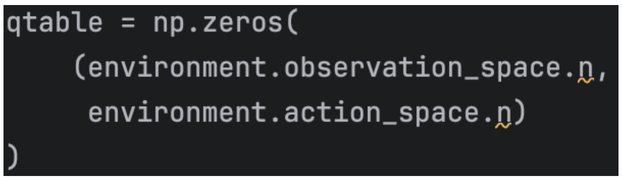
```
if rnd < epsilon:
    action = environment.action_space.sample()
# chooses best action with probability 1-epsilon
else:
    action = np.argmax(qtable[state])
```

High values of epsilon, i.e during the first stages of training, favor exploration in the form of sampling random actions to perform on each state.
As the training goes on, and as the agent learns the expected rewards, epsilon decreases and the agent is encouraged to choose actions that grant the maximum reward for a given state.

3.4 Q-Table Structure

The Q-table is a 2D table where:

- Rows represent the states.
- Columns represent the possible actions. Each entry in the table stores the expected future reward for taking a specific action in a given state.
  Over time, the table **converges**, and the agent learns the optimal actions for each state.



```
qtable = np.zeros(
    (environment.observation_space.n,
     environment.action_space.n)
)
```

It is declared as a 2-dimensional Numpy array, initialized with 0s.

3.5 Hyperparameters

For Tabular Q-Learning, the following hyperparameters were used:

- **Learning Rate (α)**: 0.1
  Controls how much the new information overrides the old, i.e how fast the algorithm converges
- **Discount Factor (γ)**: 0.99
  Controls how much future rewards are valued

- **Epsilon (ε)**: 1.0
  Initial exploration rate

- **Epsilon Decay**: 0.995
  Gradually reduces exploration as the agent learns

- **Epsilon Minimum**: 0.01
  Minimum exploration rate, to guarantee exploration also later during the training

- **Number of Episodes**: 10,000

3.6 Training Procedure

- At the beginning of training, the agent explores the environment randomly using the ϵ-greedy policy.
- Over time, as the agent updates its Q-table, it starts exploiting the learned Q-values and taking actions with the highest expected future reward.
- The Q-table converges after enough episodes, allowing the agent to follow an optimal policy.

4. Deep Q-Learning (DQL)

4.1 Overview

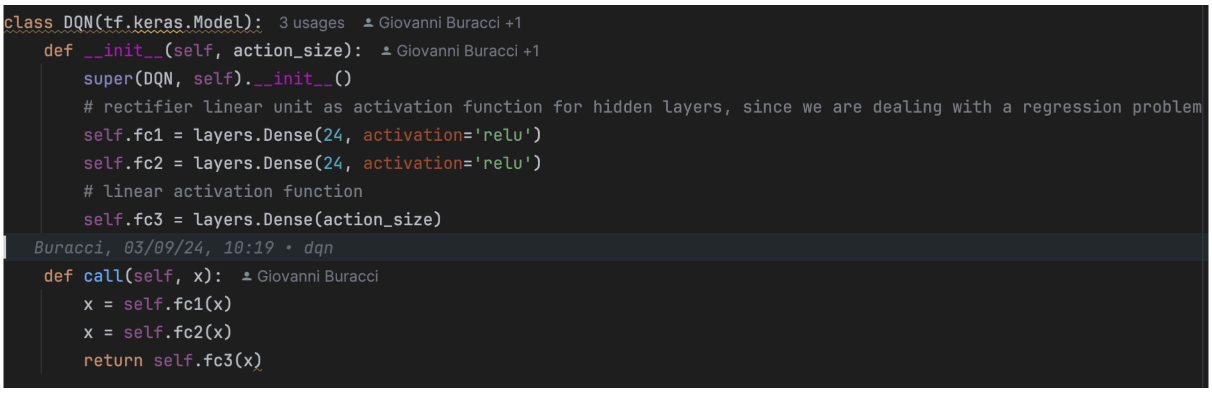
Deep Q-Learning extends Q-Learning by using a **neural network** instead of a Q-table to approximate the Q-values.
This method is especially useful when the state space is large.

4.2 DQL Architecture

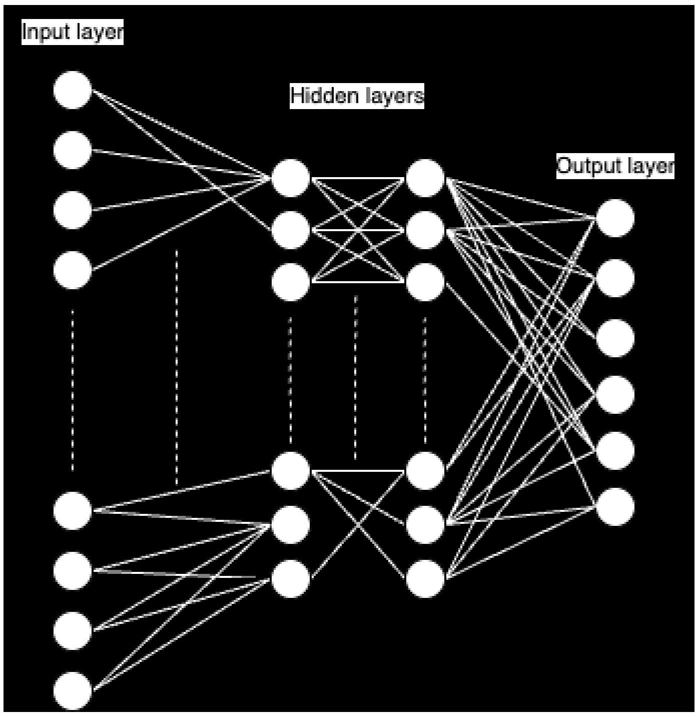
In DQL, the agent uses a neural network to approximate the Q-values for all possible actions given the current state. The architecture consists of:

- **Input Layer**: Takes in the state (one-hot encoded) as input.
- **Hidden Layers**: Two fully connected layers with ReLU activations to model complex interactions.
- **Output Layer**: Outputs the Q-values for all possible actions.

Neural Network Architecture:



```
class DQN(tf.keras.Model):  3 usages  Giovanni Buracci +1
    def __init__(self, action_size):  Giovanni Buracci +1
        super(DQN, self).__init__()
        # rectifier linear unit as activation function for hidden layers, since we are dealing with a regression problem
        self.fc1 = layers.Dense(24, activation='relu')
        self.fc2 = layers.Dense(24, activation='relu')
        # linear activation function
        self.fc3 = layers.Dense(action_size)
   Buracci, 03/09/24, 10:19 • dqn
    def call(self, x):  Giovanni Buracci
        x = self.fc1(x)
        x = self.fc2(x)
        return self.fc3(x)
```



4.3 Hyperparameters

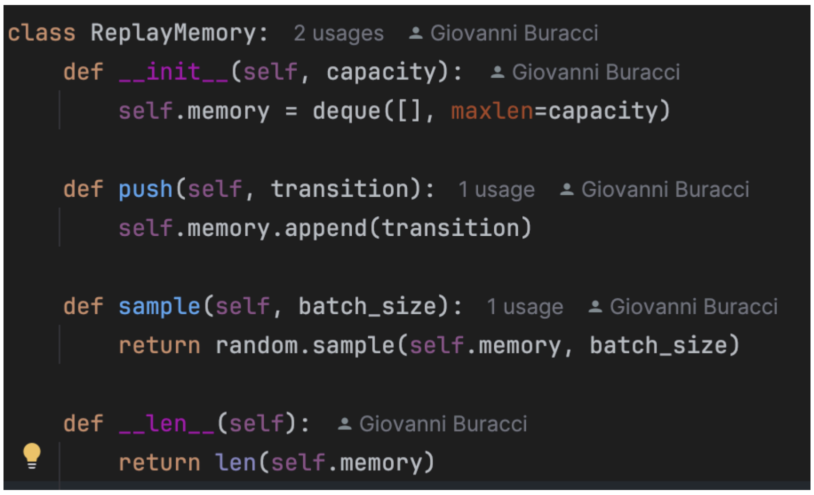
- **Learning Rate**: 0.001
- **Gamma (γ)**: 0.99
- **Epsilon (ε)**: 1.0
- **Epsilon Decay**: 0.995
- **Epsilon Minimum**: 0.01
- **Batch Size**: 64
- **Memory Capacity**: 10,000
- **Number of Episodes**: 10,000

4.4 Replay Memory

In DQL, a **replay memory** is used to store previous experiences (state, action, reward, next state, done).

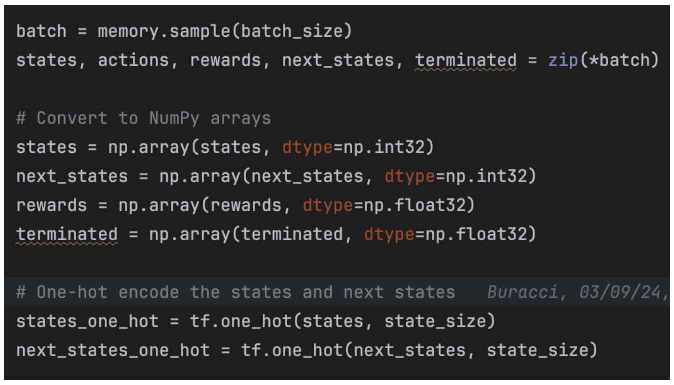
After collecting these experiences, we will store them in the replay memory and then pick them randomly (in a certain batch size) when we want to train the DQN.
This approach can give different advantages:

1. Breaks correlation between consecutive experiences
2. Stabilizes learning
3. Saves time (we don't have to perform an action and train the network for each step)

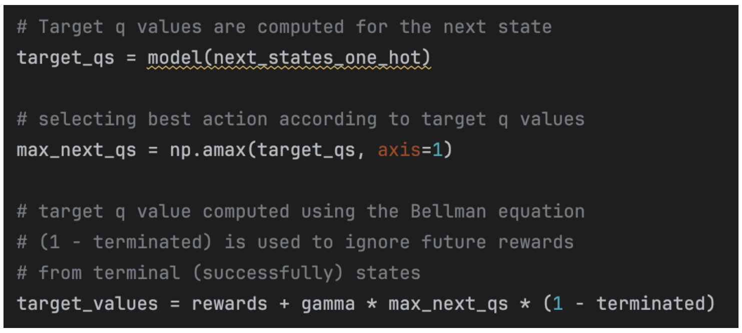

```
class ReplayMemory:  2 usages  Giovanni Buracci
    def __init__(self, capacity):  Giovanni Buracci
        self.memory = deque([], maxlen=capacity)

    def push(self, transition):  1 usage  Giovanni Buracci
        self.memory.append(transition)

    def sample(self, batch_size):  1 usage  Giovanni Buracci
        return random.sample(self.memory, batch_size)

    def __len__(self):  Giovanni Buracci
        return len(self.memory)
```

4.5 Training Process

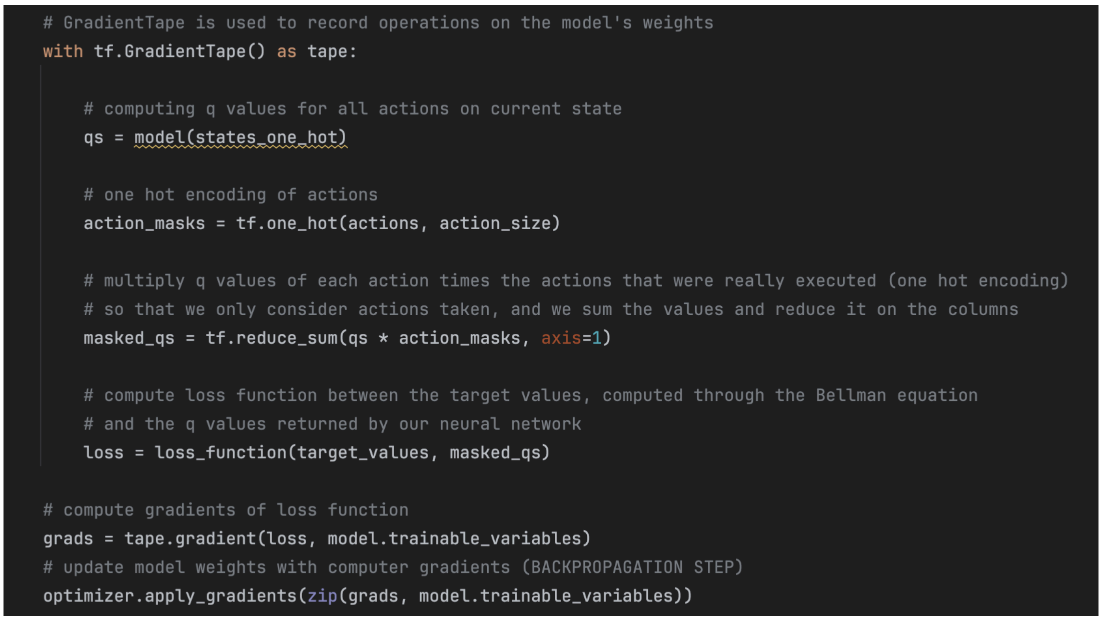
1. **Experience Replay**: Samples a batch of experiences from the replay memory and uses them to train the neural network.


```
batch = memory.sample(batch_size)
states, actions, rewards, next_states, terminated = zip(*batch)

# Convert to NumPy arrays
states = np.array(states, dtype=np.int32)
next_states = np.array(next_states, dtype=np.int32)
rewards = np.array(rewards, dtype=np.float32)
terminated = np.array(terminated, dtype=np.float32)

# One-hot encode the states and next states   Buracci, 03/09/24,
states_one_hot = tf.one_hot(states, state_size)
next_states_one_hot = tf.one_hot(next_states, state_size)
```

2. **Q-Learning Update**: Updates the target values using the Bellman equation:


```
# Target q values are computed for the next state
target_qs = model(next_states_one_hot)

# selecting best action according to target q values
max_next_qs = np.amax(target_qs, axis=1)

# target q value computed using the Bellman equation
# (1 - terminated) is used to ignore future rewards
# from terminal (successfully) states
target_values = rewards + gamma * max_next_qs * (1 - terminated)
```

3. **Gradient Descent**: The loss function, which uses Mean Squared Error (MSE), computes the difference between the predicted Q-values and the target values. Gradients are calculated using backpropagation, and the network weights are updated using gradient descent.



```python
# GradientTape is used to record operations on the model's weights
with tf.GradientTape() as tape:

    # computing q values for all actions on current state
    qs = model(states_one_hot)

    # one hot encoding of actions
    action_masks = tf.one_hot(actions, action_size)

    # multiply q values of each action times the actions that were really executed (one hot encoding)
    # so that we only consider actions taken, and we sum the values and reduce it on the columns
    masked_qs = tf.reduce_sum(qs * action_masks, axis=1)

    # compute loss function between the target values, computed through the Bellman equation
    # and the q values returned by our neural network
    loss = loss_function(target_values, masked_qs)

# compute gradients of loss function
grads = tape.gradient(loss, model.trainable_variables)
# update model weights with computer gradients (BACKPROPAGATION STEP)
optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

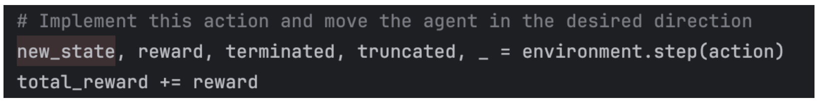
5. Experimental Results

5.1 Evaluation Metrics

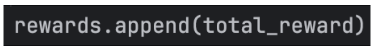
To evaluate the performance of both the Tabular Q-Learning and DQL agents, we used the following metrics:

- **Total Reward**

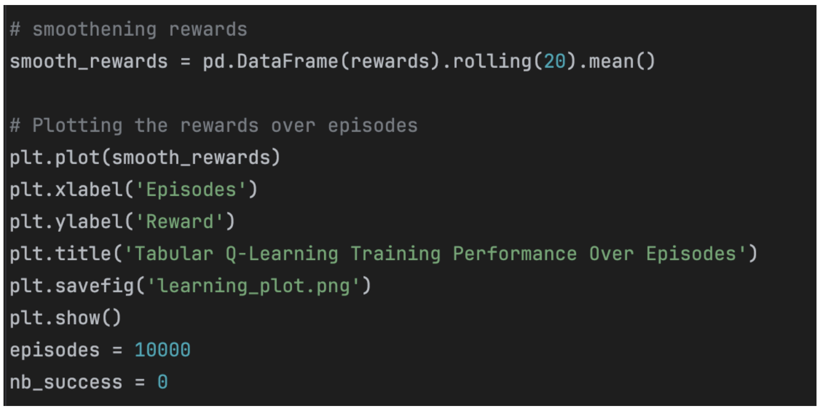
  This measure evaluates the agent's overall performance by tracking how much reward the agent accumulates in each episode.
  It helps us understand how quickly and effectively the agent is learning to reach the terminal state.



```
# Implement this action and move the agent in the desired direction
new_state, reward, terminated, truncated, _ = environment.step(action)
total_reward += reward
```



```
rewards.append(total_reward)
```

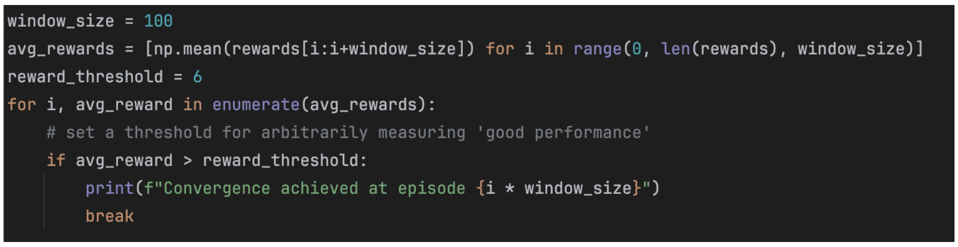


```
# smoothening rewards
smooth_rewards = pd.DataFrame(rewards).rolling(20).mean()

# Plotting the rewards over episodes
plt.plot(smooth_rewards)
plt.xlabel('Episodes')
plt.ylabel('Reward')
plt.title('Tabular Q-Learning Training Performance Over Episodes')
plt.savefig('learning_plot.png')
plt.show()
episodes = 10000
nb_success = 0
```

The rewards array has to undergo a moving average operation, otherwise it would not easily show the reward trend during training.

- **Convergence speed**

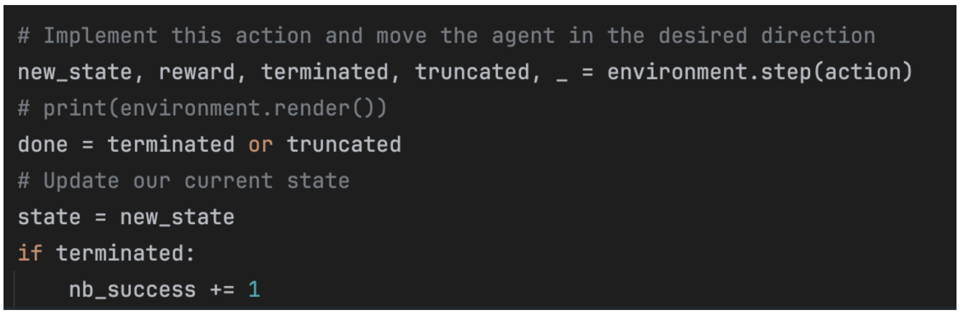
  Convergence speed measures how quickly the algorithm learns an optimal policy. In reinforcement learning, "convergence" can be defined as the point where the agent consistently achieves high (or optimal) rewards.
  Faster convergence means the method is more efficient in learning from experiences.



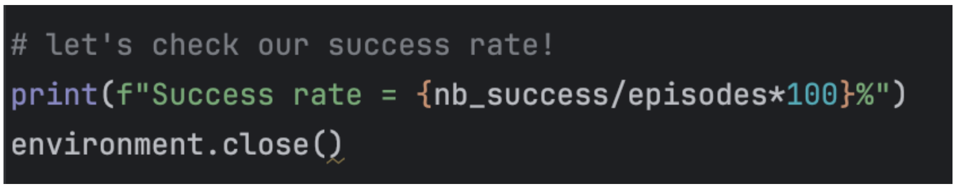
```
window_size = 100
avg_rewards = [np.mean(rewards[i:i+window_size]) for i in range(0, len(rewards), window_size)]
reward_threshold = 6
for i, avg_reward in enumerate(avg_rewards):
    # set a threshold for arbitrarily measuring 'good performance'
    if avg_reward > reward_threshold:
        print(f"Convergence achieved at episode {i * window_size}")
        break
```

- **Percentage of Episodes Solved**

  Success rate measures the percentage of episodes where the agent reaches the terminal state.
  In Taxi-v3, this means successfully picking up and dropping off the passenger.
  This metric directly evaluates how often the agent completes the task, providing insight into how robust and reliable the learned policy is.



```
# Implement this action and move the agent in the desired direction
new_state, reward, terminated, truncated, _ = environment.step(action)
# print(environment.render())
done = terminated or truncated
# Update our current state
state = new_state
if terminated:
    nb_success += 1
```

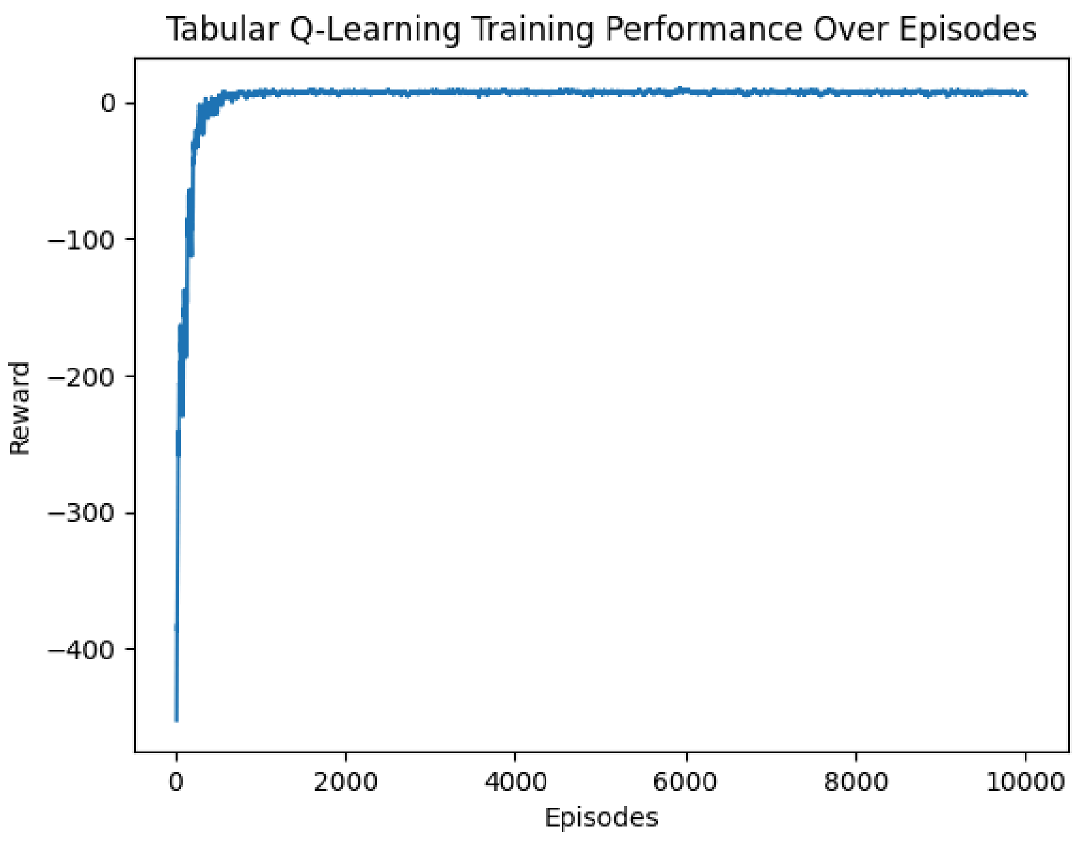


```
# let's check our success rate!
print(f"Success rate = {nb_success/episodes*100}%")
environment.close()
```

5.2 Performance Comparison

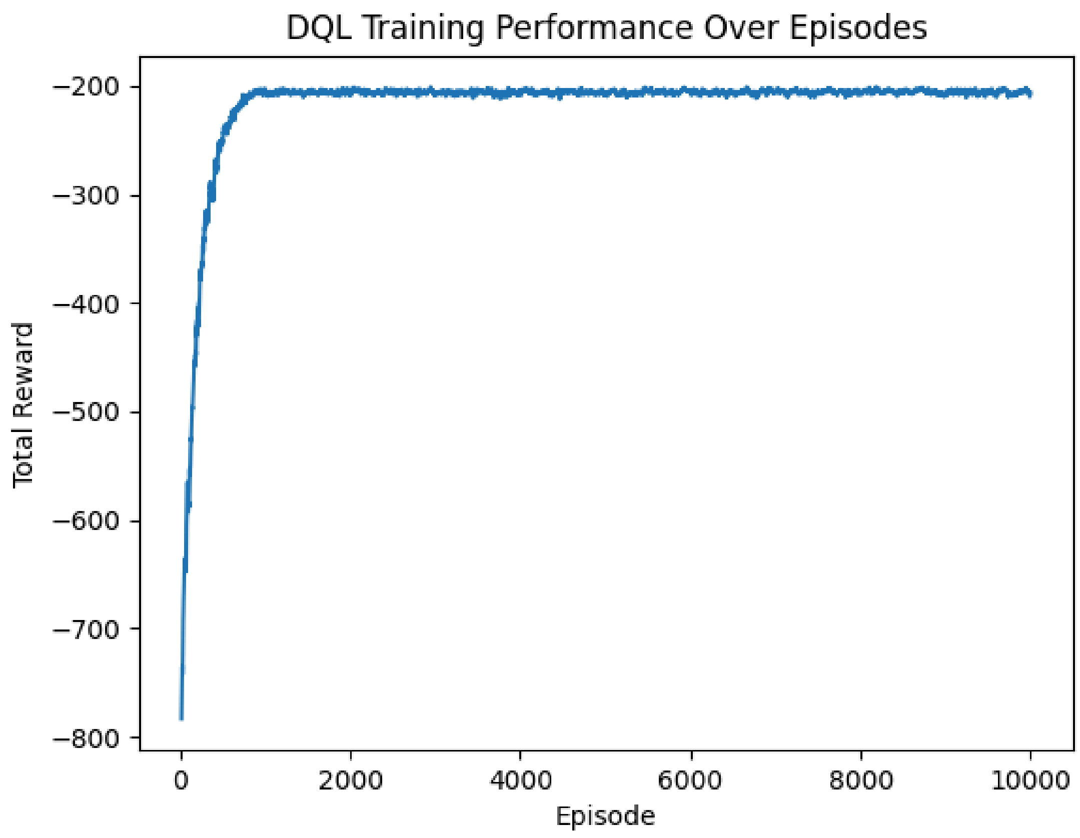
It is fundamental to notice, before comparing the performances, that the two methods were initially run using the same hyperparameters.

Tabular Q-Learning results



- Converged to an optimal policy after approximately **700 episodes**
- Achieved a **100% success rate** when playing 10000 episodes, after training for 10000 episodes.

Deep Q-Learning results



- **Failed** to converge to an optimal policy; actually, it converged to a suboptimal policy of -200 reward.
- Achieved a **0% success rate** when playing 10000 episodes, after training for 10000 episodes.

In the next section we will elaborate on the reasons why the DQL algorithm could not converge.

6. Hyperparameter Tuning

The above plot shows the DQL network reaching a suboptimal policy at around 1000 episodes, and then stabilizing at it for the rest of the training (9000 episodes).
This could possibly be due to two main reasons:

– Poor exploration vs exploitation balance

It is possible that, after 1000 episodes, the epsilon value was so low that the agent stopped exploring random actions, and instead started going more in depth on suboptimal policies with negative rewards.
Taxi, in fact, greatly penalizes negative actions, by returning -1 reward for each time step (also terminating ones) and -10 for illegal actions.

For this reason, the agent could have gotten stuck in a policy that only explored these actions and therefore could not have the time to explore an optimal one.

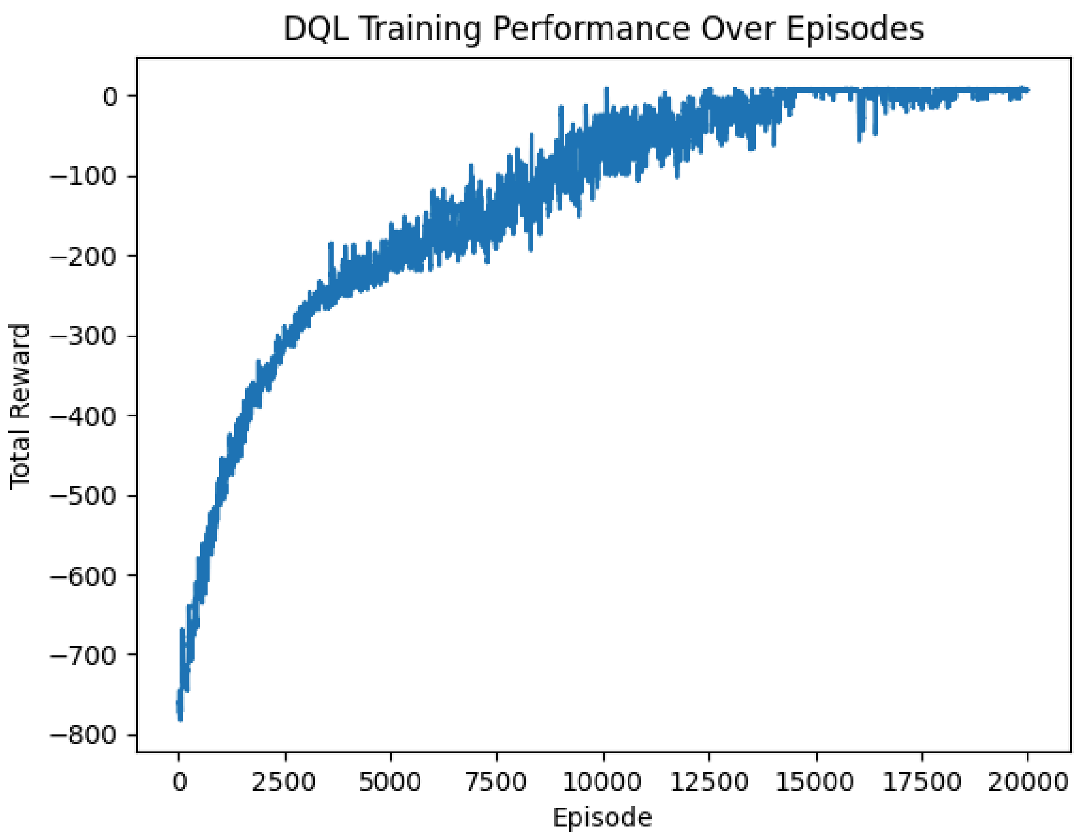
– Neural network architecture not complex enough

Even though Taxi is not a very complex environment with continuous state and action spaces, it still requires the agent to accomplish its goal by performing a precise sequence of actions.
Reaching the passenger destination, picking it up and dropping him off to the correct destination might be a hard task to learn if the network is not sufficiently expressive.

6.1 Epsilon and Exploration

- A higher initial **epsilon** allows for more exploration early in training.
- A slower **epsilon decay** encourages more exploration, which helps in environments with complex state spaces, but can slow convergence.

The first solution implemented was to raise the exploration decay value **from 0.995 to 0.9994** to allow the agent to have more time for exploration.
However, this was not enough, because 10000 episodes were not sufficient yet to learn an optimal policy.
After performing some hyperparameters tuning, finally, **20000 episodes** demonstrated to be enough for the computed policy to converge to an optimal reward.

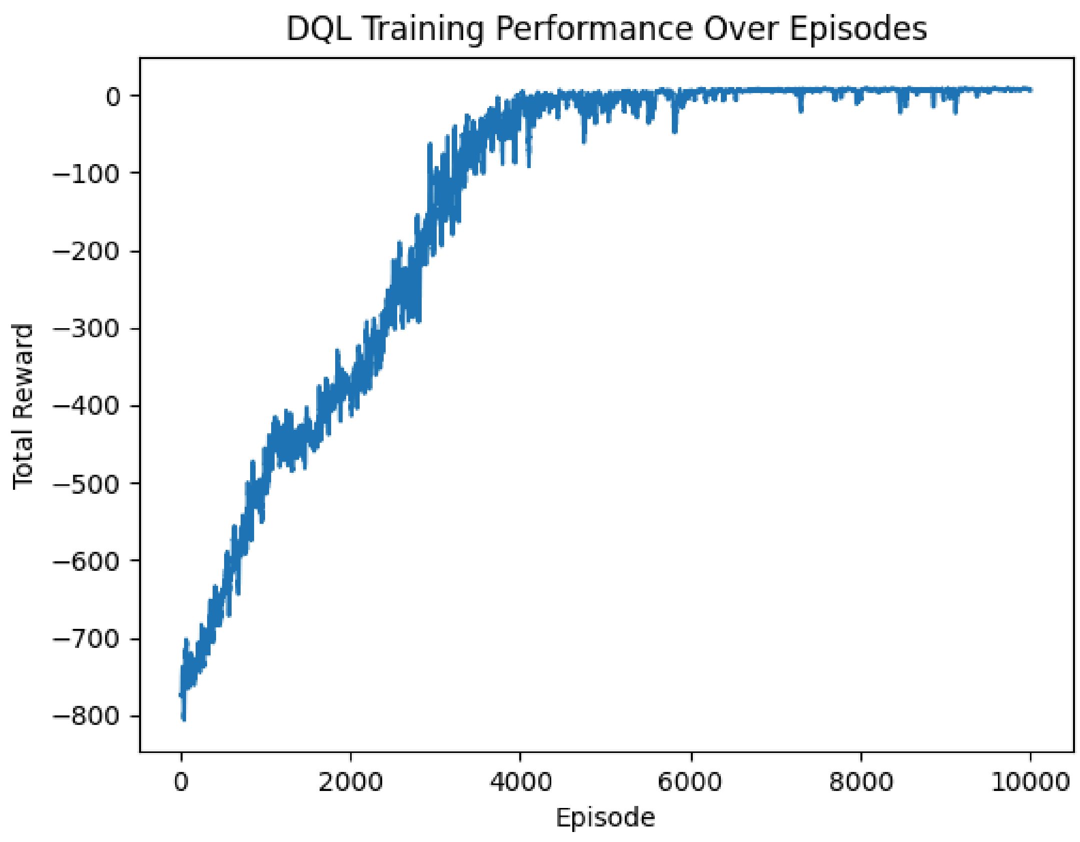


Additionally, the learning rate was halved from 0.001 to 0.0005, since we could afford to take smaller steps toward a specific policy, while obtaining more stability and a smoother learning plot.
Convergence was achieved at episode number 14600, and the policy guaranteed a 100% success rate over 10000 played episodes.

6.2 Neural network complexity

Instead of adjusting the training hyperparameters, another way to achieve an optimal policy could be to review the Neural Network organization to make it more complex.
Even though there is no precise way to know a priori how complex the network should be depending on the problem, it is possible to play with these parameters and check if they fit the solution.

To accomplish this, I added an additional hidden layer and doubled the width of the network to 48 units.
The number of episodes was kept at 10000 and also the learning rate stayed at 0.001.

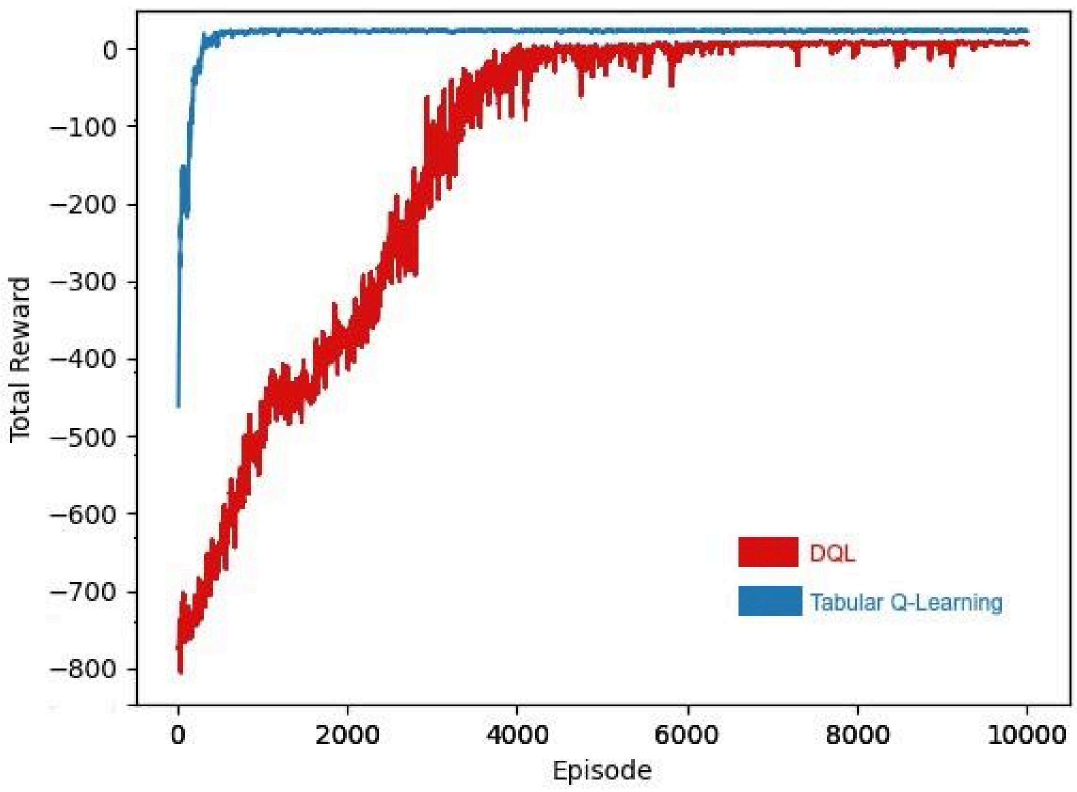


Policy convergence was achieved at episode 6700, and it granted a success rate of 100% after playing 10000 episodes.

7. Comparisons and conclusions

The plot below shows the comparison between the Tabular Q-Learning algorithm and the DQL implementation with the augmented ANN.
It is clear that not only Tabular Q-Learning converged to an optimal policy earlier, but it also remained stable through all the training.

This, in addition to the fact that it requires much less resources than training a network and requires very few tuning, makes it a more efficient method for the Taxi environment.



- **Tabular Q-Learning** is effective for small environments like Taxi-v3, where the state space is manageable.

- **DQL** is more suited for environments with larger or continuous state spaces, but it requires more computational power and careful tuning of neural network architectures and hyperparameters.

- Both methods showed strong performance on Taxi-v3 after sufficient training, with Tabular Q-Learning converging faster but DQL offering better scalability to more complex problems.

---

8. Double network DQL

In the approach we used to carry out the solution for Taxi with DQL, we used only **one network** both for:

- **Action Selection**: Choosing the action with the highest Q-value for the current state
- **Target Calculation**: Estimating the Q-value target for the next state

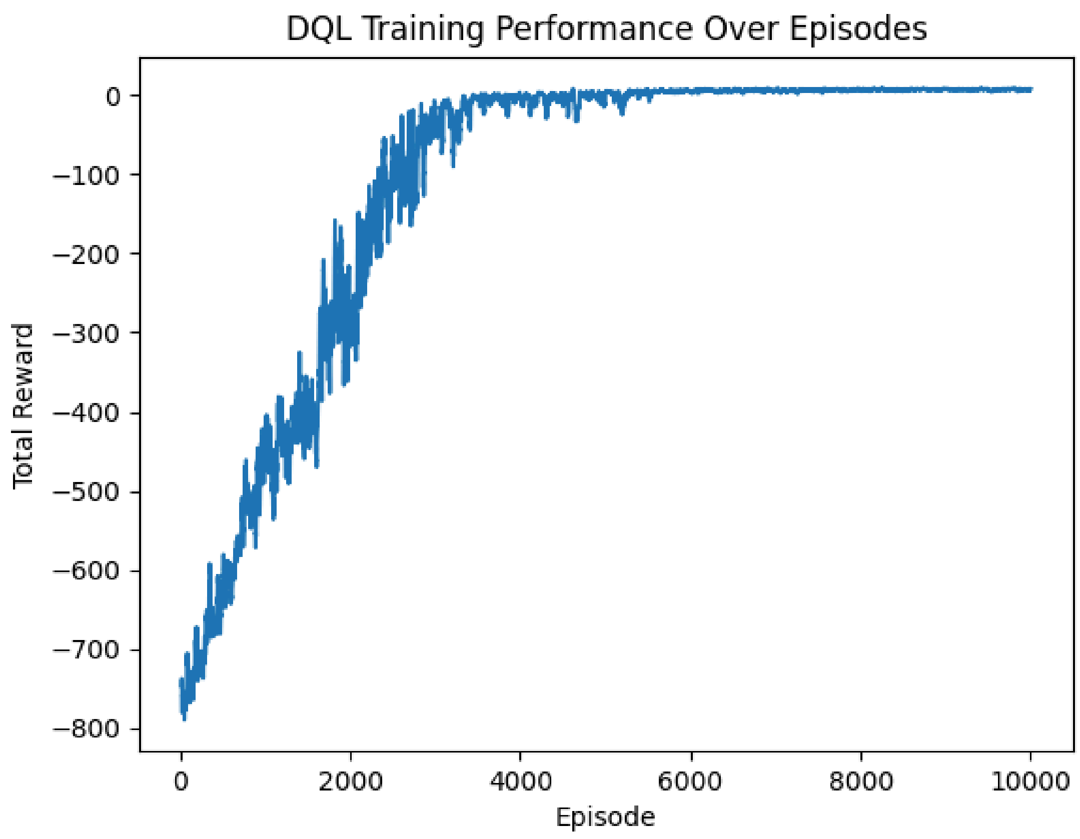
Since the same network is used for both purposes, the target Q-values are constantly shifting as the network weights are updated.
This can lead to **instability** and **slower convergence** because the target itself is moving, making it harder for the network to learn accurate Q-values.

On the other hand, it is possible to use **two separate networks**:

- **Training Network**: This network is trained every step with gradient descent. It is used to estimate the Q-values for the current state-action pairs.

- **Target Network**: This network is a copy of the training network, but it updates less frequently (i.e., every few episodes or steps). It provides more stable Q-value targets, helping avoid unstable updates caused by rapid changes in the training network.

It is generally useful to obtain more stability in training: if the Q-value targets keep changing too frequently (which happens without a target network), it can destabilize learning and lead to convergence problems.



Using a different network for the target, convergence was achieved at 6000 episodes, hence 700 episodes before the single network with an additional layer and units.