Artificial Intelligence and Machine Learning

Giuseppe Valente
Sapienza - University of Rome
Engineering in Computer Science

Contents

Chapter 1

Introduction

This work aims to study and solve Reinforcement Learning (RL) problem, using models like Q-learning and Deep Q-Network (DQN). We will introduce a basic problem called Frozen Lake, and we will study learning strategies that help the agent to solve the environment. The challenges in RL are understanding how an agent can explore and exploit an environment effectively to achieve its goal. Q-learning is a model-free RL algorithm that teaches an agent to assign values to each action it might take, conditioned on the agent being in a particular state. For any finite Markov decision process, Q-learning finds an optimal policy (maximizing the expected value of the total reward over any and all successive steps), starting from the current state. While effective for simple environments, its scalability is limited when dealing with larger or more complex state spaces. To address this limitation, DQN employs deep neural networks to approximate the Q-value function, enabling the application of RL to high-dimensional and complex environments. With this study, we will see the theoretical foundation and the practical implementation of these methods, evaluating their performance in solving the Frozen Lake problem.

Chapter 2

Frozen Lake

Frozen lake involves crossing a frozen lake from start to goal without falling into any holes by walking over the frozen lake. Frozen Lake is available into two versions: deterministic and non-deterministic setting the variable

Listing 2.1: slippery variable

```
is_slippery = True # Set the Non Deterministic environment
is_slippery = False # Set the Deterministic environment
```

In the non-determinisitc environment the agent may not always move in the intended direction while in the determinisitc environment it always moves in the intended location. In this project we use the simplest version of the Frozen Lake that is a 4x4 grid where the agent starts to position (0,0) with the goal located at the position (3,3). Holes in the ice are distributed on the cells of the grid. The agent makes moves until they reach the goal or fall in a hole. The figure 2.1 shown an example ov the Frozen Lake environment:

Figure 2.1: Frozen Lake 4x4 environment

2.1 Action Space

The agent has 4 possible actions to use:

- 0: Move left
- 1: Move down
- 2: Move right
- 3: Move up

2.2 Observation Space

The observation is an integer value representing the current position of the agent. The image in figure 2.2 shown the observation space

Figure 2.2: Frozen Lake 4x4 environment: Observation Space

2.3 Rewards

The rewards are the feedback obtain by the agent while perform an action when it is a specific state. The agent will try to maximize the rewards during the execution. In this environment the rewards are:

- Reach the goal $\rightarrow +1$
- Reach the hole $\rightarrow 0$
- Reach the frozen $\rightarrow 0$

2.4 Episode ends

The episodes ends in the following cases:

- Termination (The agent fall into hole or achieve the goal)
- Truncation (The time length of the episode, in this case is 100)

The condition of termination and truncation are given by the environment after agent takes an action

2.5 Non-Deterministic environment

We know in deterministic environement an action taken by the agent in a given state will go always in the expected destination state. This is not true for the non-deterministic environment where if the agent wants to take the action "move left" it will taken with a

$$\Pr = \frac{1}{3} \tag{2.1}$$

2.6 Brief of Q-Learning

Reinforcement Learning (RL) is one of three basic machine learning ofmachine learning concerned with how an agent should take actions in a dynamic environment to maximize a reward. Finding a balance between exploration (of uncharted environment) and exploitation (of current knowledge) with the goal of maximize the cumulative reward. The environment is typically stated in the form of a Markov decision process (MDP). The purpose of reinforcement learning is for the agent to learn an optimal policy that maximizes the reward function or other user-provided reinforcement signal that accumulates from immediate rewards.

2.6.1 Agent's learning task

A basic reinforcement learning agent interacts with its environment in discrete time steps. At each time step t, the agent receives the current state S_t and reward R_t. It then chooses an action A_t from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state S_{t+1} and the reward R_{t+1} associated with the transition (S_t,A_t,S_{t+1}) is determined. The goal of a reinforcement learning agent is to learn a policy

2.6.2 Q-Function

The Q-Function (the state-action) function is a function that helps the agent to decide which action to take in a given state in order to maximize its expected cumulative reward. In general we can write the Q-Function using the Bellman equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right) \tag{2.2}$$

Q(s, a): Current estimate of the expected cumulative reward for taking action a in state

s and following the optimal policy thereafter.
Q(s, a): The current Q-value for the state-action pair (s, a).
α**:** The *learning rate*. It controls how much the new information should override the old information. A higher value means the agent will quickly adapt to new experiences.
r: The *immediate reward* received after taking action a in state s.
γ**:** The *discount factor*. It determines how much the agent values future rewards compared to immediate rewards. A value close to 1 means the agent cares about long-term rewards, while a value close to 0 means the agent prioritizes immediate rewards.
$\max_{a'} Q(s', a')$**:** The maximum Q-value over all possible actions a' in the next state s'. This represents the best expected reward the agent can get from the next state onward.
Q(s, a): The original Q-value before the update, which is updated based on the new experience.

2.6.3 Hyperparameters

They are parameters set before the learning process begins, play a crucial role in determining the efficiency and effectiveness of the learning process.

- **Learning rate (α)**: Controls how much the Q-values are adjusted during each update. A larger learning rate leads to faster updates, but can also result in instability. A smaller learning rate may lead to slower learning.
- **Discount factor (γ)**: Determines how much the agent values future rewards compared to immediate rewards. A discount factor close to 1 means the agent values long-term rewards, while a value close to 0 means the agent focuses on immediate rewards.
- **Exploration rate (ϵ)**: In the ϵ-greedy strategy, this parameter controls the probability with which the agent chooses a random action (exploration) instead of the action with the highest Q-value (exploitation). It helps balance exploration of new actions with exploiting the known best actions.
- **Number of episodes (N)**: Specifies the number of training episodes or iterations over which the agent will interact with the environment and learn. More episodes typically lead to better learning but require more computation time.

Choosing the right set of hyperparameters is crucial for training an effective model. Poor choices of hyperparameters can lead to slow convergence, overfitting, or even complete failure of the algorithm to learn useful behavior.

2.7 Brief of Deep Q-Network (DQN)

One of the main drawbacks of Q-learning is that it becomes infeasible when dealing with large state spaces, as the size of the Q-table grows exponentially with the number of states and actions. In such cases, the algorithm becomes computationally expensive and requires a lot of memory to store the Q-values. To solve this problem we can combine Q-Learning with Deep Neural Network. The neural network receives the state as an input and outputs the Q-values for all possible actions. The following figure illustrates the difference between Q-learning and deep Q-learning in evaluating the Q-value: Essentially, deep Q-Learning

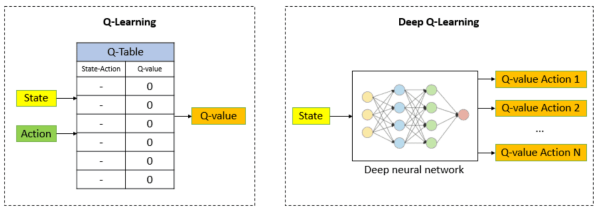

Figure 2.3: Difference between Q-Learning and DQN in evaluating Q-Values

replaces the regular Q-table with the neural network. Rather than mapping a (state, action) pair to a Q-value, the neural network maps input states to (action, Q-value) pairs. These are the steps of how DQN works:

- **Environment:** DQN interacts with an environment with a state, an action space, and a reward function. The goal of the DQN is to learn the optimal policy that maximizes cumulative rewards over time
- **Replay Memory:** DQN uses a replay memory buffer to store past experiences. Each experience is a tuple (state, action, reward, next state) representing a single transition from one state to another. The replay memory stores these experiences to sample from later randomly
- **Deep Neural Network:** DQN uses a deep neural network to estimate the Q-values for each (state, action) pair. The neural network takes the state as input and outputs the Q-value for each action. The network is trained to minimize the difference between the predicted and target Q-values.
- **Epsilon-greedy:** DQN uses the ε-greedy algorithm to balance exploration and exploitation.
- **Target Network:** DQN uses a separate target network to estimate the target Q-values. It is a copy of the main neural network with fixed parameters and it is updated periodically to prevent the overestimation of Q-values.
- **Training:** DQN trains the neural network using the Bellman equation 2.2 to estimate the optimal Q-values.The loss function is the mean squared error (MSE) between the predicted and target values. The neural network weights are updated using backpropagation and stochastic gradient descent.
- **Testing:** DQN uses the learned policy to make environmental decisions after training. The agent selects the action with the highest Q-value for a given state

In summary, DQN learns the optimal policy by using a deep neural network to estimate the Q-values, a replay memory buffer to store past experiences, and a target network to prevent overestimating Q-values. The agent uses an epsilon-greedy exploration strategy during training and selects the action with the highest Q-value during testing.

2.8 Solving the Deterministic Environment

2.8.1 Deterministic Environment

In a *deterministic* environment, the outcome of any action taken by the agent is completely predictable. For a given state s and action a, the next state s' is always the same. The environment follows strict rules, meaning that there is no randomness involved in the transition from one state to another. The reward associated with an action is also deterministic, meaning the agent always receives the same reward for performing the same action in the same state. The agent can rely on complete predictability and plan its actions with full certainty about the consequences. In deterministic environments, the Q-value can be updated directly based on the immediate reward and the known future reward, so α isn't needed. Then the update rule for the Q-function is:

$$Q(s, a) \leftarrow r + \gamma Q(s', a') \tag{2.3}$$

2.8.2 Learning function

The following algorithm shows the pseudocode of the learning function

Algorithm 1 Q-learning Algorithm (with ε-greedy)

Input: discount factor γ, exploration rate ε, number of episodes N
Initialize: Q-table $Q(s, a)$ for all states s and actions a, set episode count $i = 1$
Initialize: $\varepsilon_{min} \leftarrow 0.01$, $\varepsilon_{start} \leftarrow \varepsilon$, $k = 0.00001$
while $i \leq N$ **do**
    Initialize starting state s_1
    while not terminated or truncated s_t **do**
        Choose action a_t based on the ϵ-greedy policy:
        if random number $r \leq \epsilon$ **then**
            Select a random action a_t ▷ Exploration
        else
            Select action $a_t \leftarrow \arg\max_a Q(s_t, a)$ ▷ Exploitation
        end if
        Take action a_t, observe reward r_t and next state s_{t+1}
        $Q(s_t, a_t) \leftarrow r_t + \gamma Q(s_{t+1}, a')$ ▷ Update Q-Value
        $s_t \leftarrow s_{t+1}$
    end while
    $i \leftarrow i + 1$
    $\varepsilon \leftarrow \max\left(\varepsilon_{\min}, \varepsilon_{\text{start}} \cdot e^{-k \cdot (i+1)}\right)$
end while
Output: $\pi \leftarrow \{s : \arg\max Q[s] \mid s \in \text{range(for all states s)}\}$

Input parameters

The algorithm takes in input the discount factor (γ), the exploration rate (ϵ), the number of iterations (episodes) N.

- **Discount factor (γ)** is a value $0 \leq \gamma \leq 1$. A discount factor close to 1 means the agent values long-term rewards, while a value close to 0 means the agent focuses on immediate rewards.

- **Exploration rate** (ϵ) is a value $0 \leq \epsilon \leq 1$ It helps balance exploration of new actions with exploiting the known best actions.
- **Number of episodes** (N) is an integer value $N \geq 1$ It specifies the number of training with the agent that interacts with the environement

Q-table

In this environement instance the Q-table is represented as a matrix (2D array) composed by:

- $4x4 = 16$ rows representing the **states**
- 4 columns representing the **actions**

Then the final Q-Table is composed by $16x4 = 64$ elements In this implementation the Q-Table is zeroized.

2.8.3 Exploration and Exploitation strategy

The agent will balance the exploration and the exploitation, we need to use a very high ε parameter (closes to 1) to obtain an optimal policy.
During the iterations the algorithm will update ε value using the formula:

$$\varepsilon \leftarrow \max\left(\varepsilon_{\min}, \varepsilon_{\text{start}} \cdot e^{-k\cdot(i+1)}\right) \tag{2.4}$$

2.8.4 Output

In output the agent will return the policy learned, and the Q-Table updated.

2.8.5 Results

We learned three agents with different parameters, and collected cumulative rewards during the learning phase then the following are the results: From the 2.4 we can see:

- The green agent (with $\varepsilon = 0.1$) will not reach out the optimal policy, because the agent prefers exploitation instead of balance with exploration
- The orange agent has an high $\varepsilon = 0.98$ that allows the agent to balance exploitation and exploration. The second parameter $\gamma = 0.1$ indicates, that the agent prefers immadiate reward to future reward, and we can see the learning process start immediatly.
- The blu agent has $\varepsilon = 0.98$ that allows to balance exploitation and exploration. The second parameter $\gamma = 0.98$ indicates, that the agent prefers future rewards to immediate rewards, and we can see the learning process starts after some episodes.

We can see an highlight of the Q-Table learned by agents (blue and green agents) that help us to understand the final Q-Table learned by the two agents

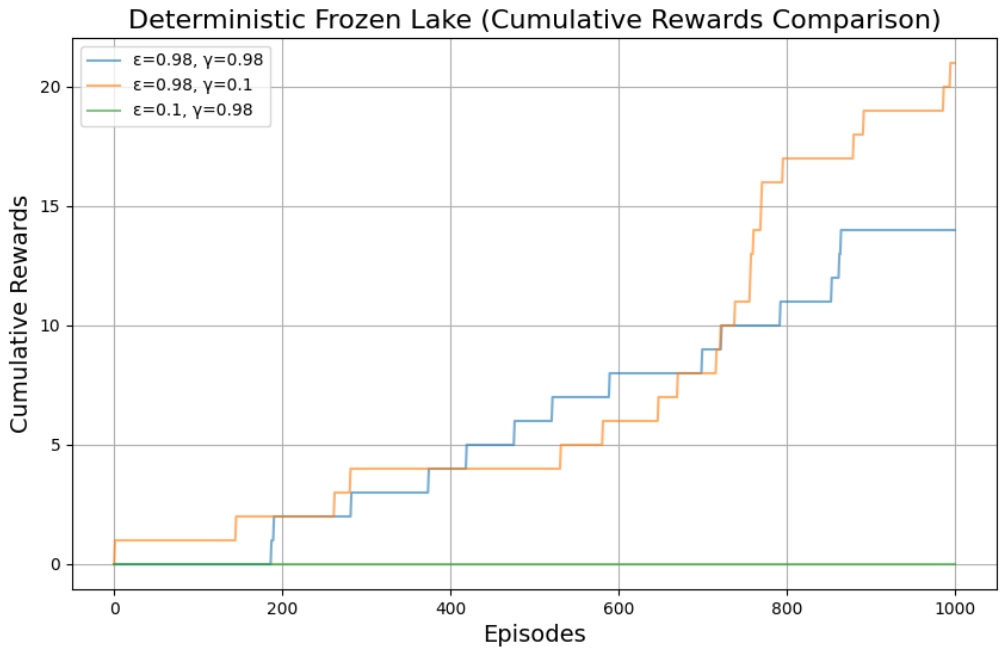

Figure 2.4: Agents comparison

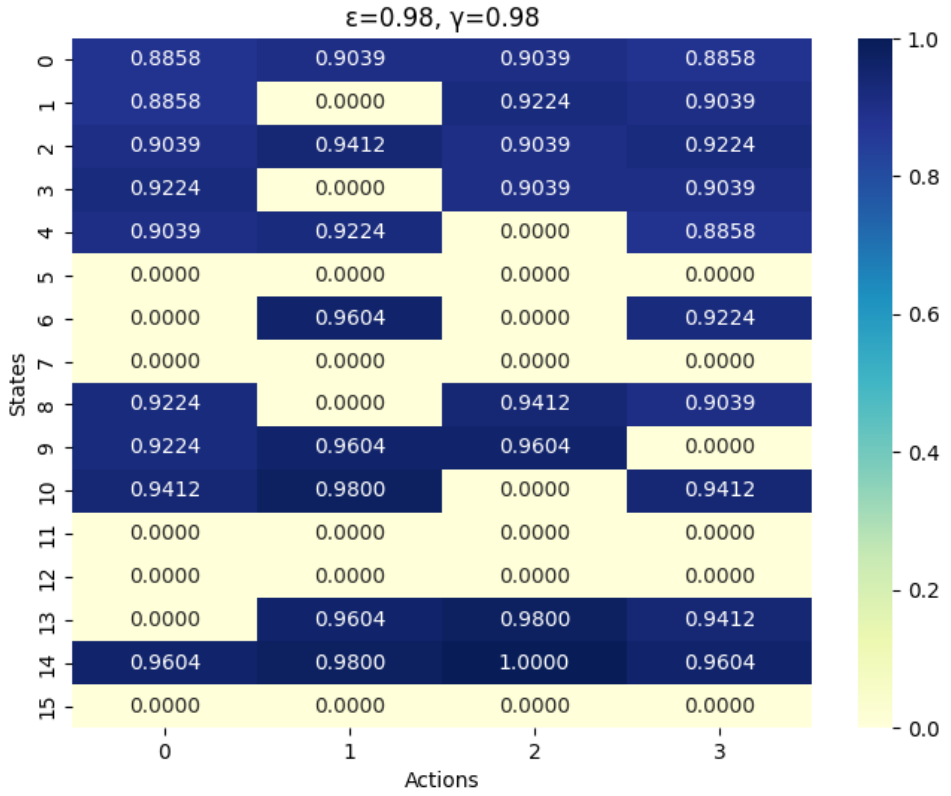

Figure 2.5: Q-Table of blue agent

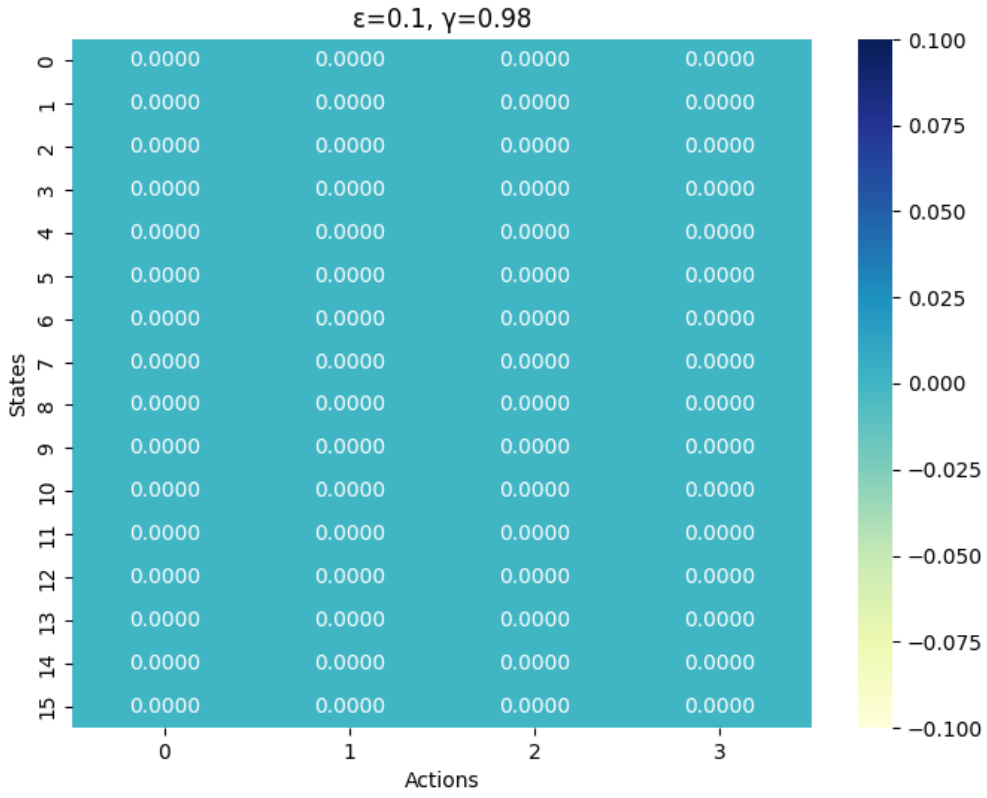

Figure 2.6: Q-Table of green agent

Using the policies learned by the agents is possible to see success rate that in a deterministic environement must be 100% In the figure 2.7 are shown the $\log(\frac{wins}{episodes})$ of the three agents:

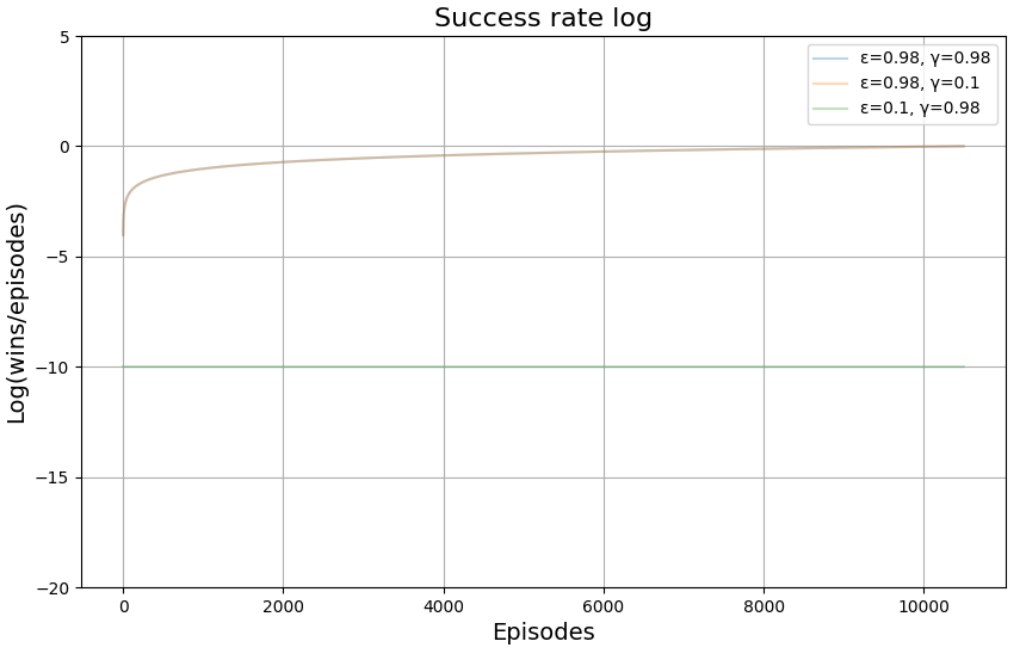

Figure 2.7: Success rate log

- The green agent has no rewards and for convenience is designed as a constant negative line, because it never converges to the optimal policy
- The other two learned agents are overlapping (you can see the brown curve that is the union of the blue and orange), and they converge to the optimal policy

2.9 Solving the Non-Deterministic Environment

2.9.1 Learning Function

The following algorithm shows how the agent is learned in a non-deterministic environment

Algorithm 2 Q-learning Algorithm (with ε-greedy)

Input: discount factor γ, exploration rate ε, learning rate α, number of episodes N
Initialize: Q-table $Q(s, a)$ for all states s and actions a, set episode count $i = 1$
Initialize: $\varepsilon_{min} \leftarrow 0.01$, $\varepsilon_{start} \leftarrow \varepsilon$, $k = 0.00005$
while $i \leq N$ **do**
    Initialize starting state s_1
    while not terminated or truncated s_t **do**
        Choose action a_t based on the ϵ-greedy policy:
        if random number $r \leq \epsilon$ **then**
            Select a random action a_t ▷ Exploration
        else
            Select action $a_t \leftarrow \arg\max_a Q(s_t, a)$ ▷ Exploitation
        end if
        Take action a_t, observe reward r_t and next state s_{t+1}
        $Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right)$ ▷ Update Q-Value
        $s_t \leftarrow s_{t+1}$
    end while
    $i \leftarrow i + 1$
    $\varepsilon \leftarrow \max\left(\varepsilon_{\min}, \varepsilon_{\text{start}} \cdot e^{-k \cdot (i+1)}\right)$
end while
Output: $\pi \leftarrow \{s : \arg\max Q[s] \mid s \in \text{range(for all states s)}\}$

Difference

In non-deterministic environment we have another input parameter the learning rate. It controls how much the Q-values are adjusted during each up-date. A larger learning rate leads to faster updates, but can also result in instability.

- α closes to 0 the learning rate is very slow, but the Q-Table will have stable values.
- α closes to 1 the learning rate is fast, but the Q-Table will have unstable values.

Another important difference is the Q-Function used, that in this case is 2.2

2.9.2 Results

Now we can see the results of three learned agents.

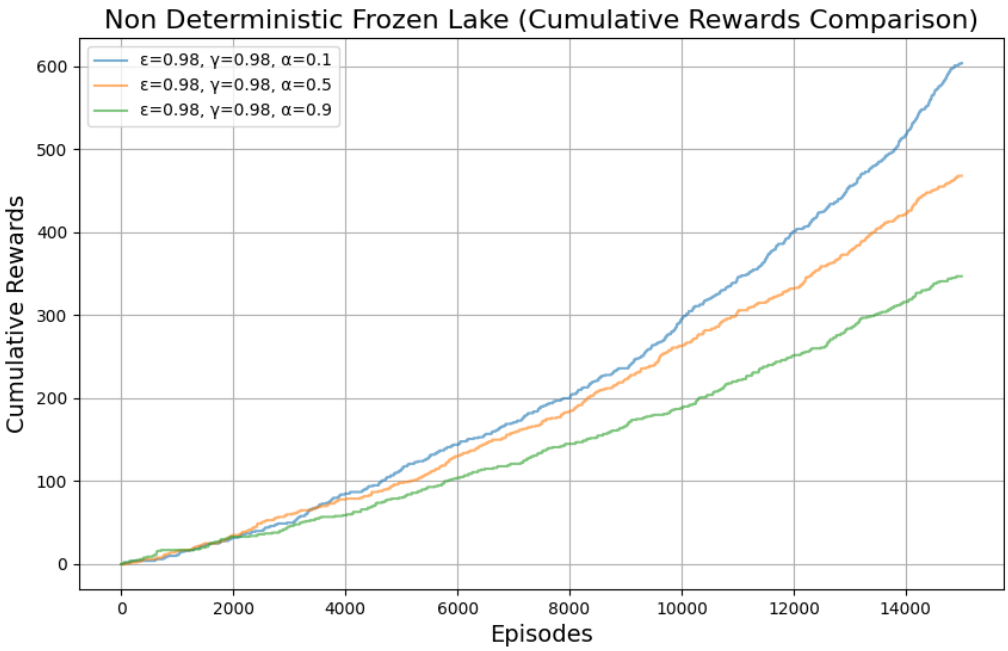

Figure 2.8: Agents comparison

In this analysis we are interesting to understand the learning rate parameter and how it changes the learning curve of the agents.

- The blu agent has $\alpha = 0.1$, this means that the agent will update with caution the new Q-Values, this causes a slow initial grows, but after a specific point, it will converge most speedly then each other to the convergence.
- The orange agent has $\alpha = 0.5$, and it represent the balanced situation between the two curves, it will increase most speedly at begin of the blue agent, but it will converge slowly the the blue agent, but speedly then the green agent
- The green agent has $\alpha = 0.9$, it will learn speedly to the begin, but this lead instability then it will reach the optimal policy most slowley then the other curves.

In the following picture are shown the Q-Table learned by the three agents:

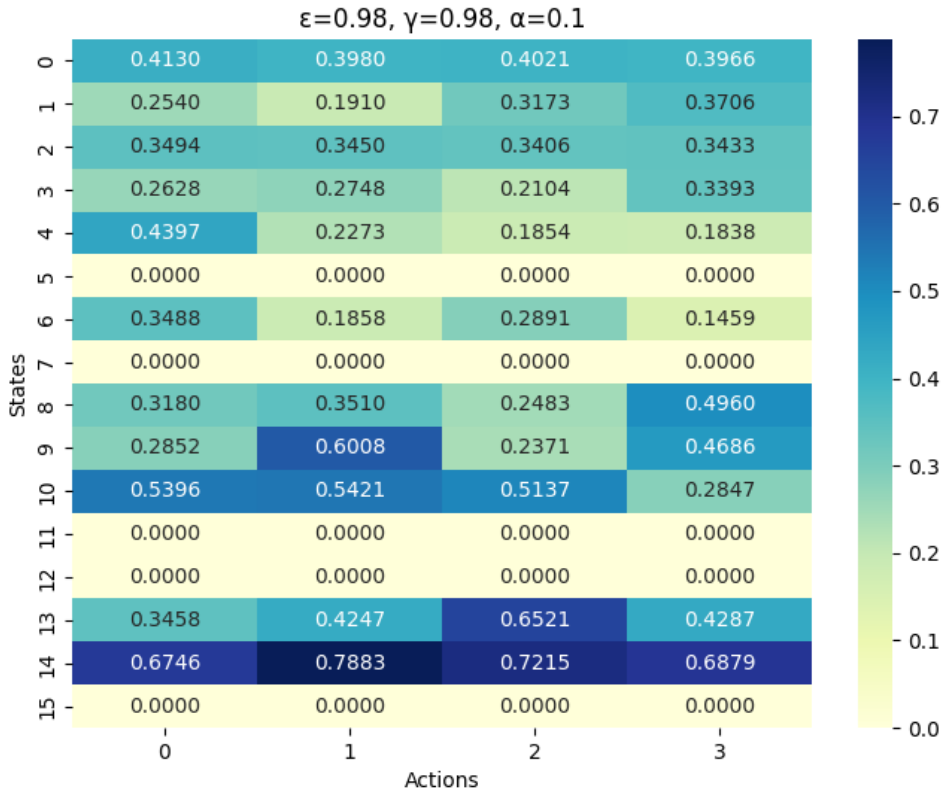

Figure 2.9: Q-Table of blue agent

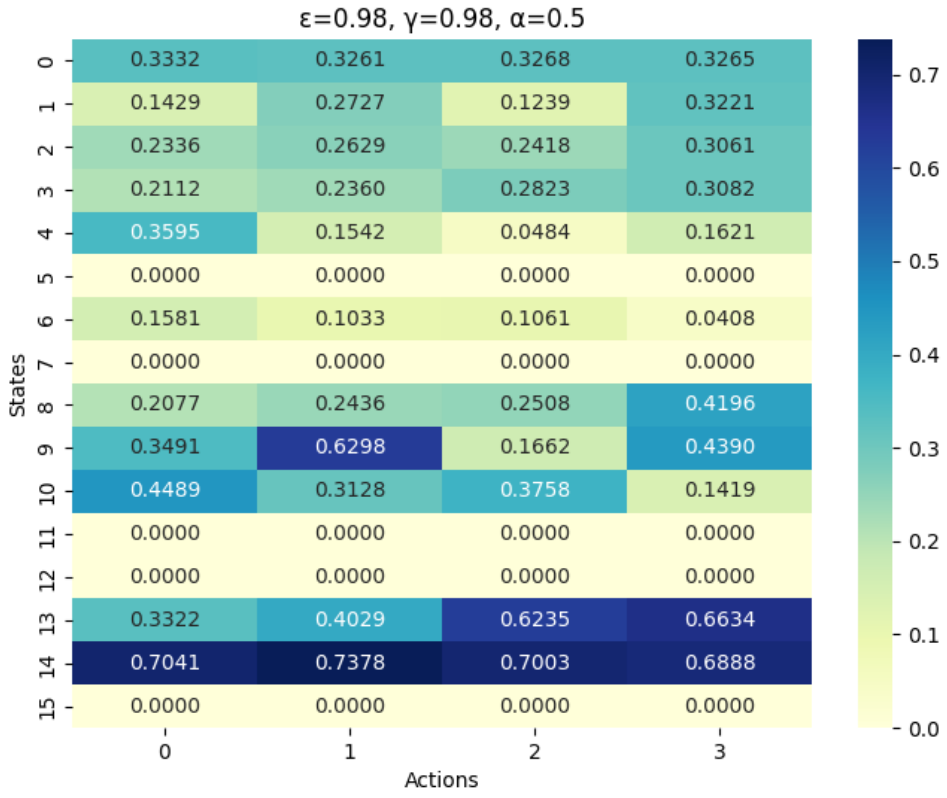

Figure 2.10: Q-Table of orange agent

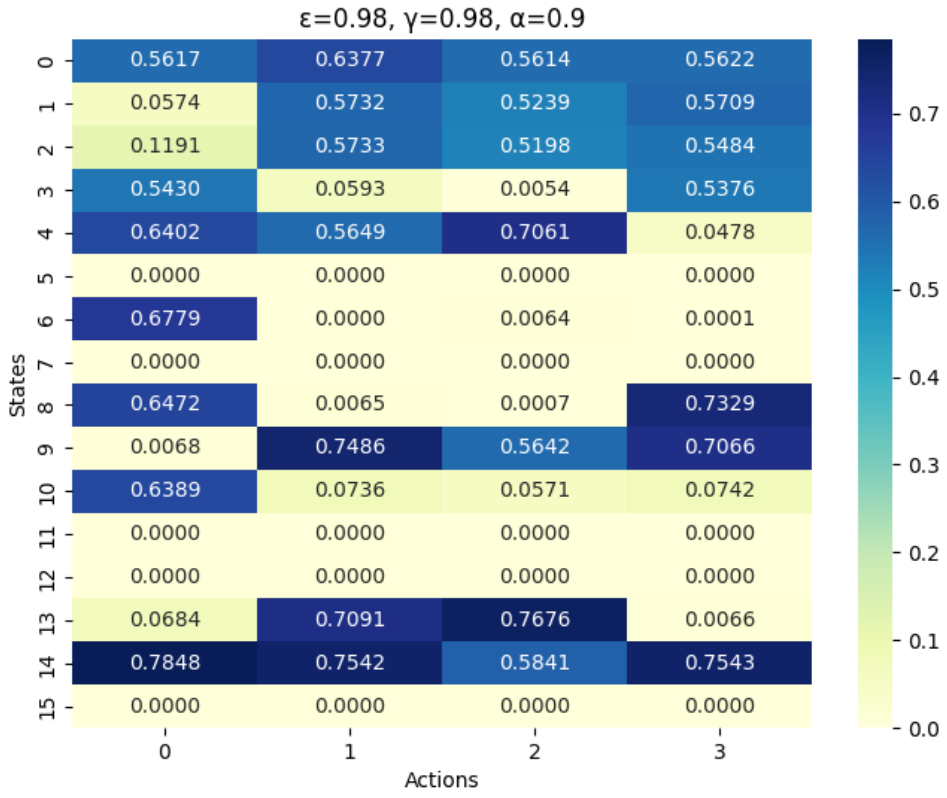

Figure 2.11: Q-Table of green agent

The obtained result end with the measure of the success rate. In the following figure is shown the overlap success rate of three agents. A very important fact is the agents learned an optimal policy, they do not have a success rate of 100%.

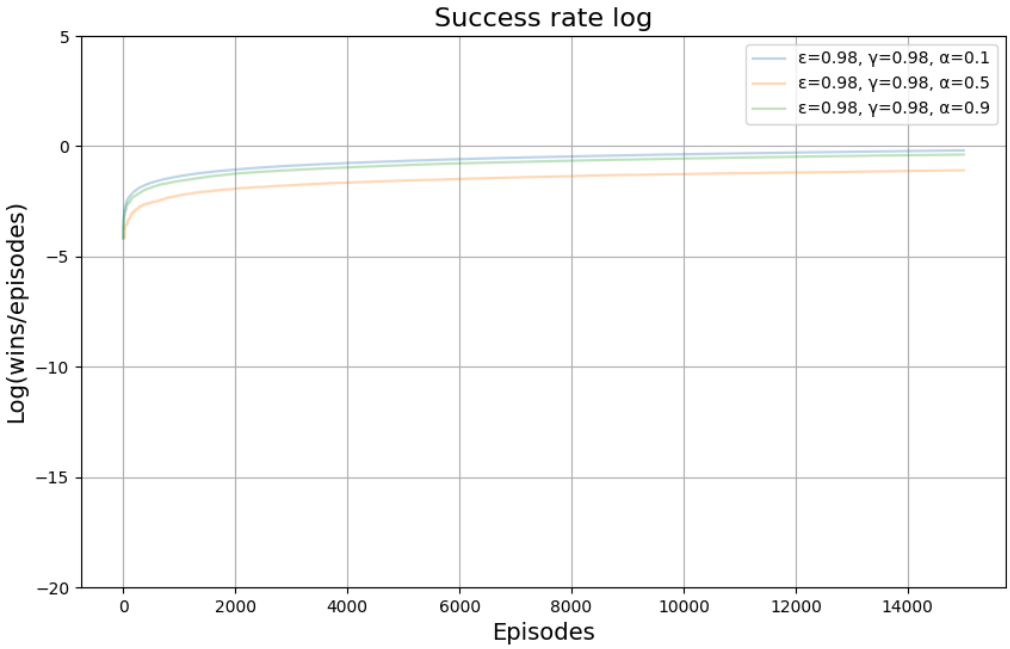

Figure 2.12: Success rate log

2.10 Solving the Frozen Lake using DQN

In our implementation we used the canonical classes:

- DQN class to represent the DQN implementation
- The ReplayMemory class to store the agent's experiences

2.10.1 DQN class details

Is very important understand how the DQN class is implemented because it defines the Neural Network architecture. For this reason we share the code of the class

Listing 2.2: DQN Neural Network Implementation

```
class DQN(nn.Module):
    def __init__(self, in_states, h1_nodes, out_actions):
        super().__init__()

        self.fc1 = nn.Linear(in_states, h1_nodes)
        self.out = nn.Linear(h1_nodes, out_actions)
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.out(x)
        return x
```

- **fc1:** Is the first fully connected layer (between the input states and the hidden layer)

- **out:** Is the second fully connected layer (between the hidden layer and the out actions)
- **forward():** This method defines the data flow through the network. This is how the network processes input data to produce an output.
- `self.fc1(x)`:
  - The input $\mathbf{x}$ is passed through the first fully connected layer (`fc1`).
  - The layer performs a linear transformation:

$$\mathbf{x}_{\text{out}} = \mathbf{x}_{\text{in}} \cdot \mathbf{W}^T + \mathbf{b},$$

    where:
    - $\mathbf{x}_{\text{in}}$ is the input vector,
    - $\mathbf{W}$ is the weight matrix of the layer
    - $\mathbf{b}$ is the bias term.
- `F.relu(...)`:
  - The output from `fc1` is passed through a **ReLU activation function**.
  - ReLU introduces non-linearity into the model by applying the function:

$$\text{ReLU}(z) = \max(0, z),$$

    which ensures that negative values are set to 0, while positive values remain unchanged.

2.10.2 ReplayMemory

It is queue implementation to store the agent's experiences. It has a capacity, and when it is full the oldest data is automatically deleted (FIFO policy). Offers the methods to insert and get the values from the buffer and a method **sample()** During training, batches of transitions are sampled randomly from the memory.

2.10.3 FrozenLakeDQN

This is the main class responsible for training and testing the agent in the FrozenLake environment. It uses the following parameters:

2.10.4 Hyperparameters

- **alpha:** Learning rate for the optimizer (Adam).
- **gamma:** Discount factor for future rewards.
- **epsilon:** For epsilon-greedy exploration (the probability of taking a random action).
- **network_sync_rate:** How often to sync the target network with the policy network.

- **replay_memory_size:** Maximum size of the experience replay buffer.
- **mini_batch_size:** Number of experiences sampled for training.
- **Loss Function and Optimizer:** Initializes the loss function (Mean Squared Error) and optimizer (None initially; later set to Adam).

2.10.5 Train function pseudocode

The DQN architecture used in this method is shown in the figure 2.13

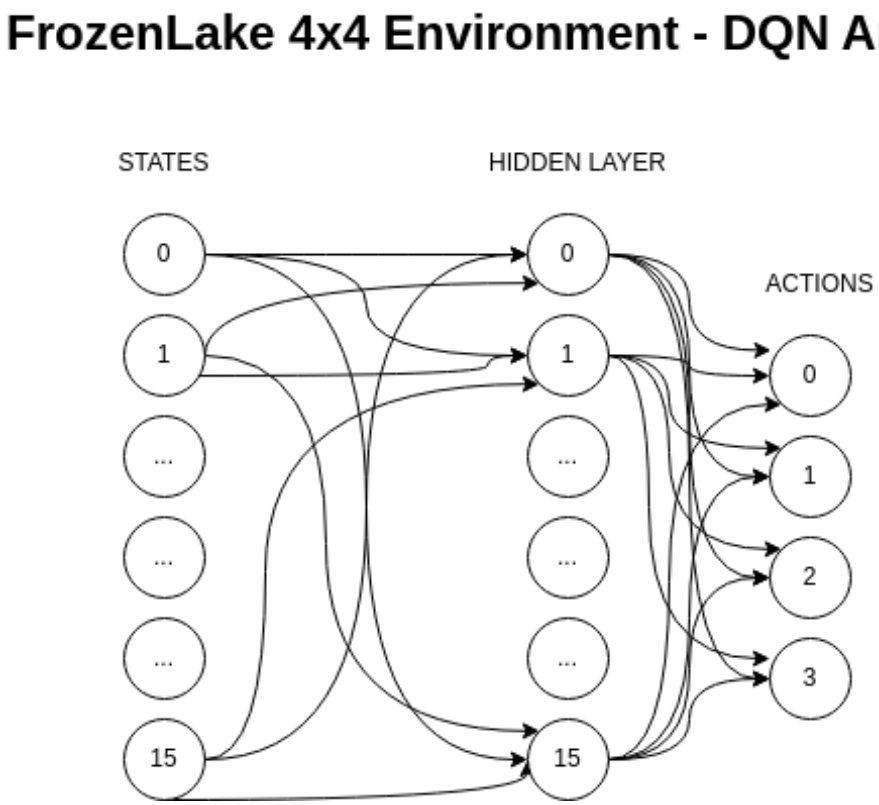

Figure 2.13: DQN architecture for ForzenLake 4x4

Algorithm 3 Train method for Deep Q-Network (DQN)

Input: Number of episodes *episodes*, render flag, slippery flag
Initialize environment *env* with FrozenLake-v1 (4x4 grid)
num_states ← number of states in environment (16 states for 4x4 grid)
num_actions ← number of actions in environment (4: left, down, right, up)
Initialize replay memory *memory* with size *replay_memory_size*
Initialize policy network *policy_dqn* with input size *num_states* and output size *num_actions*
Initialize target network *target_dqn* as a copy of *policy_dqn*
Initialize optimizer *optimizer* (Adam for *policy_dqn* with learning rate α)
Initialize *rewards_per_episode* ← zeros array of size *episodes*
Initialize *epsilon_history* ← empty list
step_count ← 0
for $i = 1$ **to** *episodes* **do**
    Reset environment and get initial state *state*
    terminated ← False, *truncated* ← False
    while *terminated* is False and *truncated* is False **do**
        if random() ¡ ϵ **then**
            Select random action *action* from *env.action_space*
        else
            Select best action *action* ← $\arg\max Q(state)$ using *policy_dqn*
        end if
        Execute action *action* and observe $new\_state, reward, terminated, truncated$
        Store experience $(state, action, new\_state, reward, terminated)$ in memory
        state ← *new_state*
        Increment *step_count*
    end while
    if $reward = 1$ **then**
        $rewards\_per\_episode[i] \leftarrow 1$
    end if
    if $len(memory) > mini\_batch\_size$ **then**
        Sample a mini-batch from *memory*
        Optimize policy network using the mini-batch
    end if
    Decay epsilon: $\epsilon \leftarrow \max(\epsilon - \frac{1}{episodes}, 0)$
    Append ϵ to *epsilon_history*
    if $step\_count > network\_sync\_rate$ **then**
        Synchronize target network: $target\_dqn.load\_state\_dict(policy\_dqn.state\_dict())$
        Reset *step_count* ← 0
    end if
end for
Close environment
Return $policy\_dqn, rewards\_per\_episode, epsilon\_history$

2.10.6 Optimize method pseudocode

Algorithm 4 Optimize method for Deep Q-Network (DQN)

Input: mini-batch of experiences, *policy_dqn*, *target_dqn*
Initialize *current_q_list* ← empty list
Initialize *target_q_list* ← empty list
for each experience $(state, action, new\_state, reward, terminated)$ in mini-batch **do**
    if *terminated* is True **then**
        Set *target* ← *reward*
    else
        $target \leftarrow reward + \gamma \cdot \max Q(new\_state)$ using *target_dqn*
    end if
    Get *current_q* ← *policy_dqn*(*state*)
    Append *current_q* to *current_q_list*
    Get *target_q* ← *target_dqn*(*state*)
    Adjust *target_q*[*action*] ← *target*
    Append adjusted *target_q* to *target_q_list*
end for
Compute *loss* ← Mean Squared Error between *current_q_list* and *target_q_list*
Zero gradients in *optimizer*
Perform backpropagation on *loss*
Update *policy_dqn* using the *optimizer*

The above algorithms help us to solve the Frozen Lake problem and we will try to explain how they work.

2.10.7 The train method

The train method handles training process of the agent environment.

Initialization

- Environment:
  - spaces ← 16
  - actions ← 4
- Replay Memory to store agent's experiences
- Neural Networks
  - Policy DQN → NN used by agent to select actions based on the current state
  - Target DQN → A copy of Policy Network used to compute stable Q-values for training.
- The optimizer (Adam) is used to update the weights of the policy network based on the loss computed during training.

Main Loop

The loop runs for a specified number of episodes (episodes), and in each episode, the agent interacts with the environment to improve its policy. Each episode starts with the environment being reset, and the agent begins at a specific state until it achieve the goal (or timing terminated) or fall into an hole. The train method uses the epsilon-greedy strategy to balance exploration and exploitation:

- ε-greedy:
  - With probability ε, it selects a random action (exploration)
  - With probability $1 - \varepsilon$ it selects the best action based on the policy network (exploitation).
- Experience storage
- The agent stores the current state, action, next state, reward in the replay memory.
- ε-decay Epsilon is decayed over time, which reduces exploration and increases exploitation as training progresses.
- Once enough experiences are stored in memory, a mini-batch of experiences is sampled and passed to the optimize method to update the policy network.
- The parameters of the target network are periodically updated to match the policy network after a certain number of steps. This helps improve stability during training by ensuring that the target Q-values do not change too frequently.

2.10.8 The optimize method

This method is responsible for updating the policy network based on the experiences sampled from memory. The Q-value update is based on the Bellman equation:

- If the episode is terminated, the Q-value is set to the immediate reward (0 or 1).
- Otherwise, the target Q-value is computed with the Bellman equation.
- Loss: is computed as difference between the current Q-value predicted by the policy network and the target Q-value computed using the target network
- Backpropagation: the optimize updates the weights of the policy network to minimize the loss

2.10.9 State DQN Input conversion

This method converts the state (an integer) into one-hot encoded vector to use as input of DQN For example if the state is 1 and the environement has 16 states, the input tensor will be an array of 16 elements with a 1 at the index 1

2.10.10 Results

An interesting comparison could be performed between a Q-Learning agent and a DQN agent. To make this comparison we build two environment Frozen Lake one with 16 states, and the second with 64 states.

2.10.11 Comparison in non-determinisitc 4x4 Environment

The figures 2.14 and 2.15 shown the comparison of the learning curve and success rate between the Q-Learning agent and the DQN agent in a non-determinisitc environement In the both picture we can see the DQN has better perfomcance in both cases.

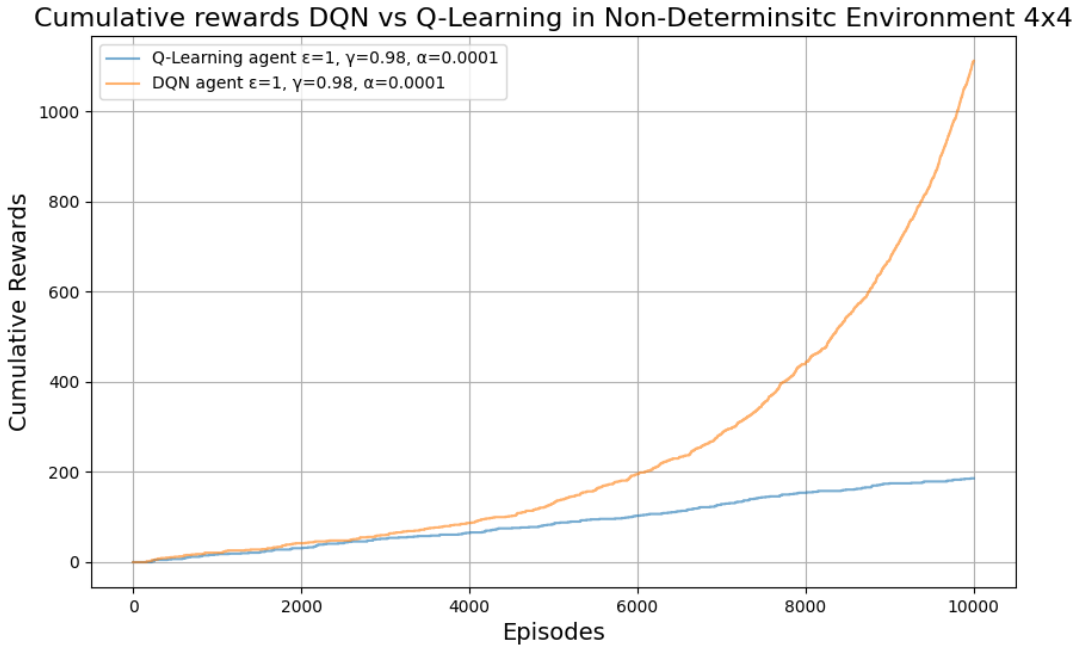

Figure 2.14: Learning curve DQN vs Q-Learning in a non deterministic environement

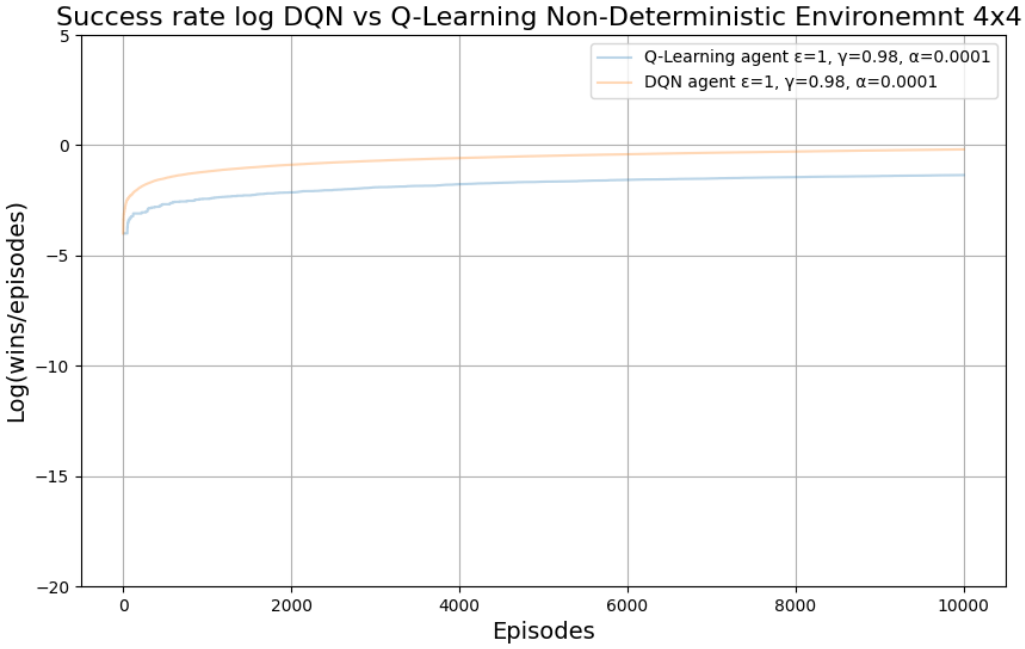

Figure 2.15: Success rate DQN vs Q-Learning in a non deterministic environement

2.11 Conclusion

The Frozen Lake problem is as a basic example in RL, providing an environment to explore both traditional Q-learning and DQN. Q-learning, a tabular method, is efficient for environments with discrete and relatively small state-action spaces. With its Q-table allows it to efficiently learn an optimal policy with limited computational overhead. However, it is impractical for problems with large or continuous state spaces, where the Q-table would grow exponentially. DQN uses neural networks to approximate Q-values, enabling them to generalize across vast or continuous state spaces. The choice between Q-learning and DQN depends by the environment and the available computational resources. For smaller, well-defined problems like Frozen Lake, Q-learning is often sufficient. However, as environments increase in complexity, DQNs provide a robust framework to tackle challenges that traditional methods cannot handle.