

# Table of Contents

- [1 Objective](#)
- ▼ [2 Introduction for Q-Learning](#)
  - [2.1 What is Q-Learning?](#)
  - [2.2 Q-Function](#)
  - [2.3 Q-table](#)
  - ▼ [2.4 The Q-Learning algorithm](#)
    - [2.4.1 Step 1: We initialize the Q-Table](#)
    - [2.4.2 Step 2: Choose action using Epsilon Greedy Strategy.](#)
    - [2.4.3 Step 3: Perform action  \$A\_t\$ , gets reward  \$R\_{t+1}\$  and next state  \$S\_{t+1}\$](#)
    - [2.4.4 Step 4: Update  \$Q\(S\_t, A\_t\)\$](#)
  - [2.5 Off-policy vs On-policy](#)
- ▼ [3 Q-Learning tutorials](#)
  - ▼ [3.1 Q-Learning with FrozenLake-v1](#)
    - [3.1.1 Install dependencies](#)
    - [3.1.2 Import the packages](#)
  - ▼ [3.2 Create Frozen Lake environment](#)
    - [3.2.1 Understanding the FrozenLake environment](#)
    - [3.2.2 Create and Initialize the Q-table](#)
    - [3.2.3 Define the epsilon-greedy policy](#)
    - [3.2.4 Define the hyperparameters](#)
    - [3.2.5 Training the model](#)
    - [3.2.6 Trained Q-Learning table](#)
    - [3.2.7 Model evaluation](#)
  - [3.3 Visualizing the results](#)
- ▼ [4 LAB Assignment](#)
  - [4.1 Exercise \(Q-Learning with Taxi-v3 🚖\) \(100 Points\)](#)
  - [4.2 Step 0 Import the packages](#)
  - [4.3 Step 1 Create Taxi-v3 🚖 environment](#)
  - [4.4 Step 2 Create the Q-table and initialize it](#)
  - [4.5 Step 3 Configure the hyperparameters](#)
  - [4.6 Step 4 Q Learning algorithm](#)
  - [4.7 Step 5 Model evaluation](#)
  - [4.8 Step 6 Visualizing the results](#)

## LAB14 tutorial for Machine Learning Q-Learning

The document description are designed by Jia Yanhong in 2022. Nov. 23th

## 1 Objective

- Understand the theory of Q-Learning
- Be able to code from scratch a Q-Learning agent.

- Be able to use **Gym**, the environment library.
- Complete the LAB assignment and submit it to BB or sakai.

## 2 Introduction for Q-Learning

### 2.1 What is Q-Learning?

Q-Learning is an **off-policy value-based method that uses a TD approach to train its action-value function:**

- **Off-policy:** we'll talk about that at the end of this chapter.
- **Value-based method:** finds the optimal policy indirectly by training a value or action-value function that will tell us the value of each state or each state-action pair.
- **Uses a TD approach:** updates its action-value function at each step instead of at the end of the episode.

### 2.2 Q-Function

**Q-Learning is the algorithm we use to train our Q-Function**, an **action-value function** that determines the value of being at a particular state and taking a specific action at that state.

$$Q^\pi(s_t, a_t) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | s_t, a_t]$$

Q-Values for the state given a particular state      Expected discounted cumulative reward      Given the state and action

Given a state and action, our Q Function outputs a state-action value (also called Q-value)

The **Q** comes from "the Quality" of that action at that state.

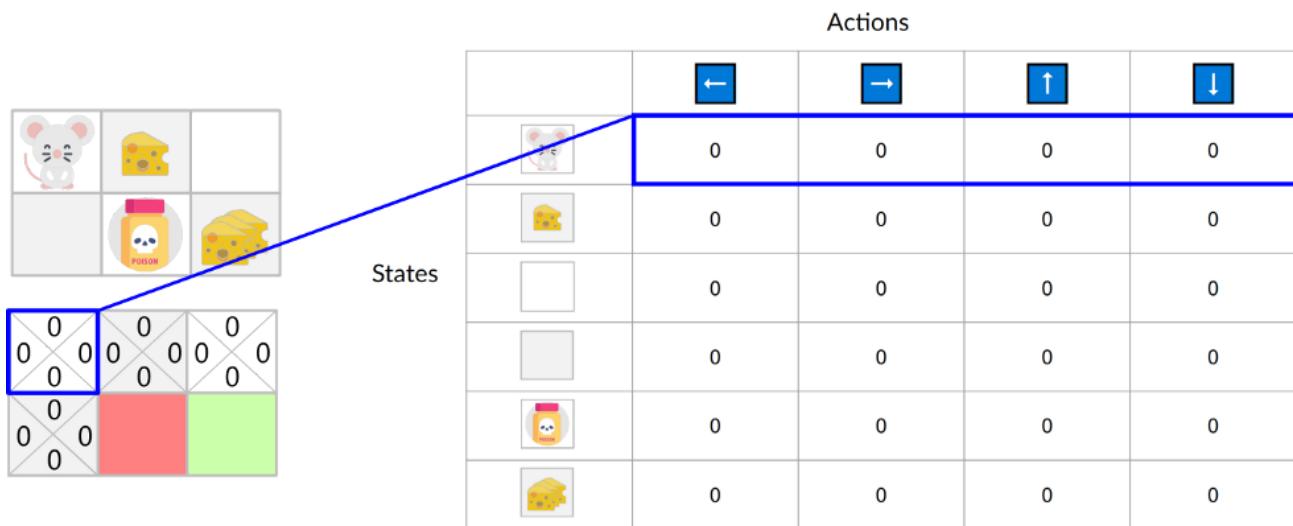
### 2.3 Q-table

Internally, our Q-function has a **Q-table**, a table where each cell corresponds to a state-action value pair **value**. Think of this Q-table as the **memory or cheat sheet of our Q-function**.

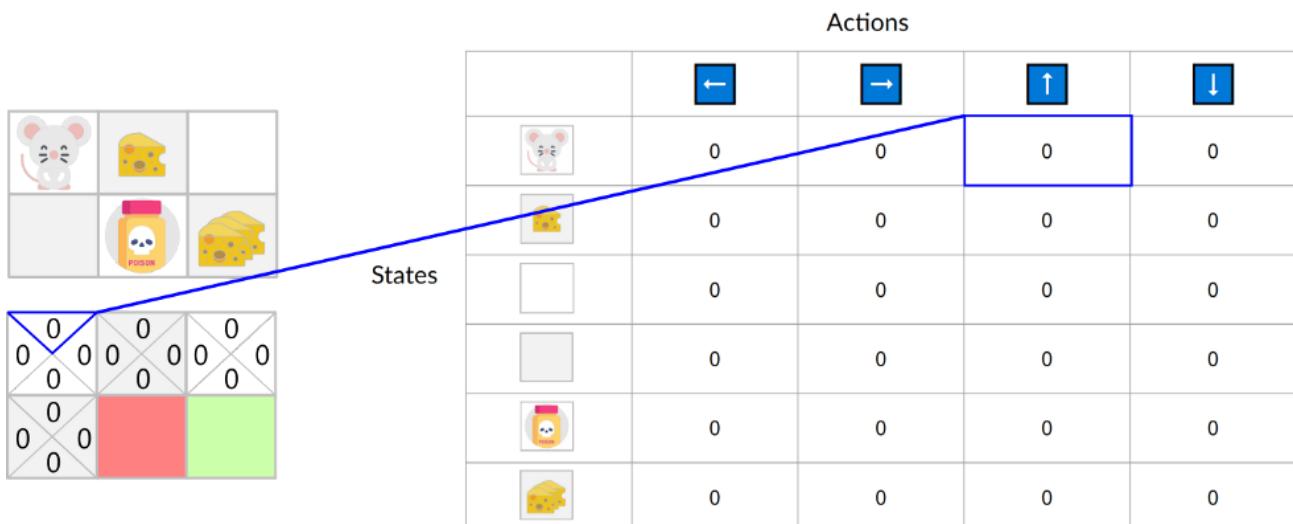
If we take this maze example:



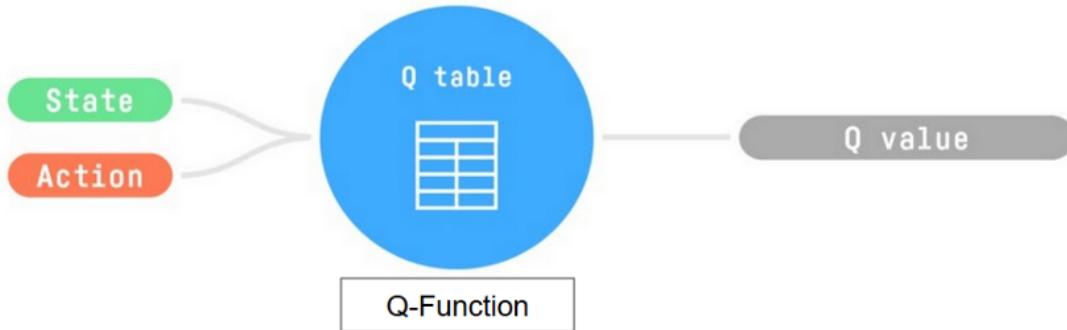
The Q-Table is initialized. That's why all values are = 0. This table **contains, for each state, the four state-action values.**



Here we see that the **state-action value of the initial state and going up is 0:**



Therefore, Q-function contains a Q-table **that has the value of each-state action pair**. And given a state and action, **our Q-Function will search inside its Q-table to output the value**.



Given a state and action pair, our Q-function will search inside its Q-table to output the state-action pair value (the Q value).

If we recap, **Q-Learning is the RL algorithm that:**

- Trains **Q-Function** (an **action-value function**) which internally is a **Q-table that contains all the state-action pair values**.
- Given a state and action, our Q-Function **will search into its Q-table the corresponding value**.
- When the training is done, **we have an optimal Q-function, which means we have optimal Q-Table**.
- And if we **have an optimal Q-function**, **we have an optimal policy** since we know for each state what is the best action to take.

## The link between Value and Policy:

$$\pi^*(s) = \arg \max_a Q^*(s, a)$$

Finding an optimal value function leads to having an optimal policy.

But, in the beginning, **our Q-Table is useless since it gives arbitrary values for each state-action pair** (most of the time, we initialize the Q-Table to 0 values). But, as we'll **explore the environment and update our Q-Table, it will give us better and better approximations**.

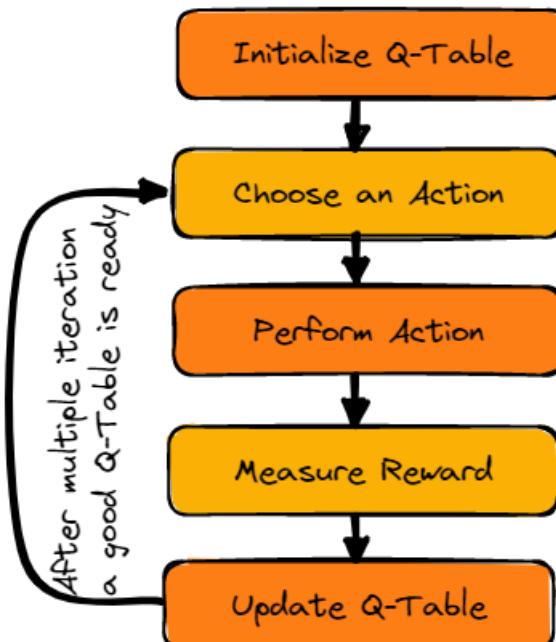


We see here that with the training, our Q-Table is better since, thanks to it, we can know the value of each state-action pair.

So now that we understand what Q-Learning, Q-Function, and Q-Table are, **let's dive deeper into the Q-Learning algorithm.**

## 2.4 The Q-Learning algorithm

The Q-Learning algorithm flow is as follows:



Here is the Q-Learning pseudocode:

**Algorithm 14:** Sarsamax (Q-Learning)

---

**Input:** policy  $\pi$ , positive integer  $num\_episodes$ , small positive fraction  $\alpha$ , GLIE  $\{\epsilon_i\}$   
**Output:** value function  $Q$  ( $\approx q_\pi$  if  $num\_episodes$  is large enough)  
Initialize  $Q$  arbitrarily (e.g.,  $Q(s, a) = 0$  for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}(s)$ , and  $Q(\text{terminal-state}, \cdot) = 0$ )  
**for**  $i \leftarrow 1$  **to**  $num\_episodes$  **do** ↗ Step 1  
   $\epsilon \leftarrow \epsilon_i$   
  Observe  $S_0$   
   $t \leftarrow 0$   
  **repeat**  
    Choose action  $A_t$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy) Step 2  
    Take action  $A_t$  and observe  $R_{t+1}, S_{t+1}$  Step 3  
     $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$  Step 4  
     $t \leftarrow t + 1$   
  **until**  $S_t$  is terminal;  
**end**  
**return**  $Q$

**2.4.1 Step 1: We initialize the Q-Table**

## Q-Learning, Step 1

Initialize  $Q$  arbitrarily (e.g.,  $Q(s, a) = 0$  for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}(s)$ , and  $Q(\text{terminal-state}, \cdot) = 0$ )

	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

We initialize the Q-Table

We need to initialize the Q-Table for each state-action pair. **Most of the time, we initialize with values of 0.**

**2.4.2 Step 2: Choose action using Epsilon Greedy Strategy**

## Q-Learning, Step 2

Choose action  $A_t$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)



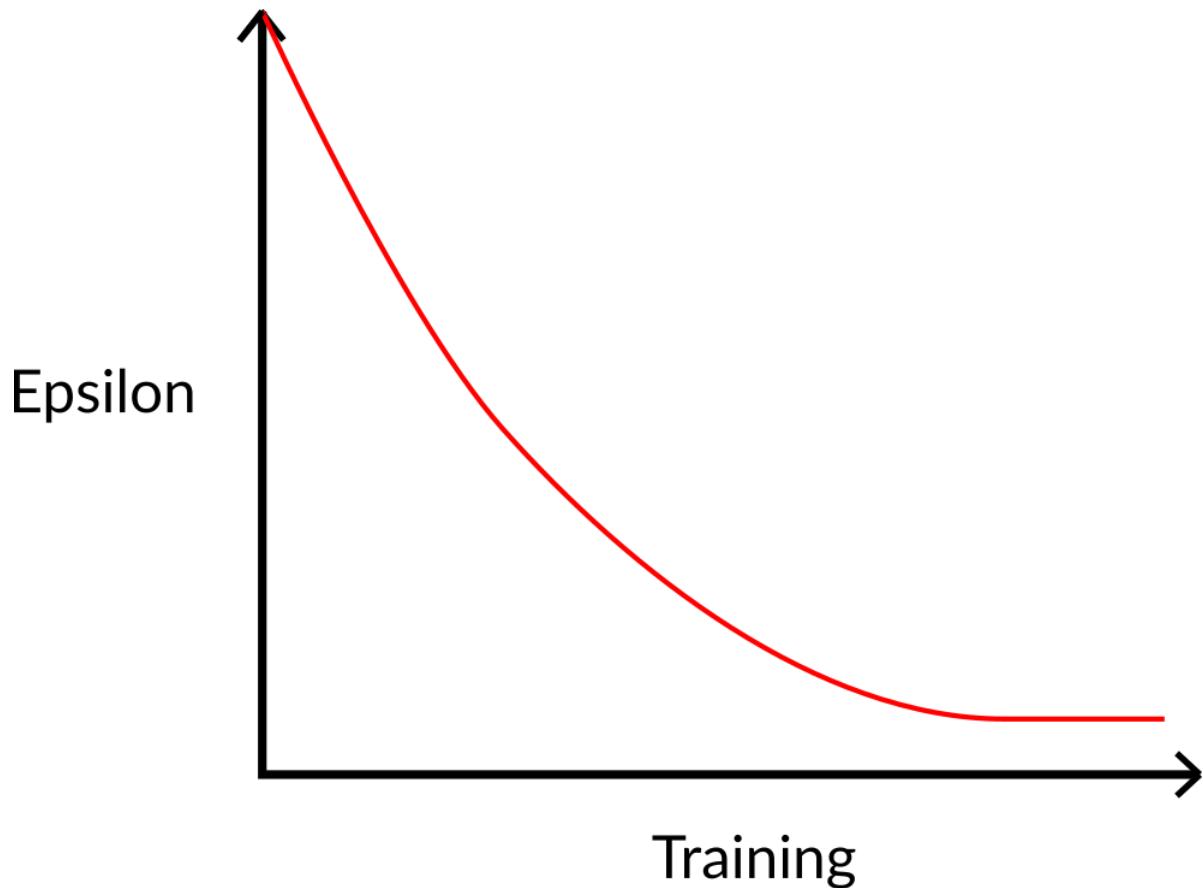
Choose the action using  $\epsilon$ -greedy policy

Epsilon Greedy Strategy is a policy that handles the exploration/exploitation trade-off.

The idea is that we define epsilon  $\epsilon = 1.0$ :

- *With probability  $1 - \epsilon$*  : we do **exploitation** (aka our agent selects the action with the highest state-action pair value).
- With probability  $\epsilon$ : we do **exploration** (trying random action).

At the beginning of the training, **the probability of doing exploration will be huge since  $\epsilon$  is very high, so most of the time, we'll explore**. But as the training goes on, and consequently our **Q-Table gets better and better in its estimations, we progressively reduce the epsilon value** since we will need less and less exploration and more exploitation.



#### 2.4.3 Step 3: Perform action $A_t$ , gets reward $R_{t+1}$ and next state $S_{t+1}$

### Q-Learning, Step 3

Take action  $A_t$  and observe  $R_{t+1}, S_{t+1}$

#### 2.4.4 Step 4: Update $Q(S_t, A_t)$

Remember that in TD Learning, we update our policy or value function (depending on the RL method we choose) **after one step of the interaction**.

To produce our TD target, **we used the immediate reward  $R_{t+1} * R_{t+1}$  plus the discounted value of the next state best state-action pair\*\* (we call that bootstrap).**

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

New value of state t	Former estimation of value of state t	Learning Rate	Reward	Discounted value of next state
----------------------	---------------------------------------	---------------	--------	--------------------------------

TD Target

Therefore, our  $Q(S_t, A_t)$  update formula goes like this:



To get the **best next-state-action pair value**, we use a greedy policy to select the next best action.

**Note that this is not an epsilon greedy policy, this will always take the action with the highest state-action value.**

**It's why we say that this is an off-policy algorithm.**

## 2.5 Off-policy vs On-policy

### Off-policy vs On-policy

- **Off-policy:** using a different policy for acting and for updating.

Choose action  $A_t$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy) | Epsilon Greedy Policy  
Take action  $A_t$  and observe  $R_{t+1}, S_{t+1}$

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

Greedy Policy

- **On-policy:** using the same policy for acting and updating.

Choose action  $A_0$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

$t \leftarrow 0$

**repeat**

  | Take action  $A_t$  and observe  $R_{t+1}, S_{t+1}$

  | Choose action  $A_{t+1}$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

  | 
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

Epsilon Greedy Policy

## 3 Q-Learning tutorials

Two Q-Learning tutorials, one for us to do together and one for Assignment. The gym environment of the two tutorials is as follows: 🎮 Environments:

- [FrozenLake-v1 \(\[https://www.gymlibrary.ml/environments/toy\\\_text/frozen\\\_lake/\]\(https://www.gymlibrary.ml/environments/toy\_text/frozen\_lake/\)\)](https://www.gymlibrary.ml/environments/toy_text/frozen_lake/)
- [Taxi-v3 \(\[https://www.gymlibrary.ml/environments/toy\\\_text/taxi/\]\(https://www.gymlibrary.ml/environments/toy\_text/taxi/\)\)](https://www.gymlibrary.ml/environments/toy_text/taxi/)

## 3.1 Q-Learning with FrozenLake-v1

### 3.1.1 Install dependencies

In [75]:

```
1 # %%capture
2 # ! pip install gym==0.24
3 # ! pip install pygame
4 # ! pip install numpy
5 # ! pip install imageio imageio_ffmpeg
```

### 3.1.2 Import the packages

In [2]:

```
1 import numpy as np
2 import gym
3 import random
4 import imageio
5 import tqdm
```

Warning: Gym version v0.24.0 has a number of critical issues with `gym.make` such that at the `reset` and `step` functions are called before returning the environment. It is recommended to downgrading to v0.23.1 or upgrading to v0.25.1

## 3.2 Create Frozen Lake environment

We're going to train our Q-Learning agent **to navigate from the starting state (S) to the goal state (G) by walking only on frozen tiles (F) and avoid holes (H)**.

We can have two sizes of environment:

- map\_name="4x4" : a 4x4 grid version
- map\_name="8x8" : a 8x8 grid version

The environment has two modes:

- is\_slippery=False : The agent always move in the intended direction due to the non-slippery nature of the frozen lake.
- is\_slippery=True : The agent may not always move in the intended direction due to the slippery nature of the frozen lake (stochastic).

You can also custom your own grid using:

```
desc=["SFFF", "FHFH", "FFFH", "HFFG"]
gym.make('FrozenLake-v1', desc=desc, is_slippery=True)
```

but we'll use the default environment for now.

In [3]:

```
1 # Create the FrozenLake-v1 environment using 4x4 map and non-slippery version
2 env = gym.make("FrozenLake-v1", map_name="4x4", is_slippery=False)
```

### 3.2.1 Understanding the FrozenLake environment

In [78]:

```
1 print("____OBSERVATION SPACE____ \n")
2 print("Observation Space", env.observation_space)
3 print("Sample observation", env.observation_space.sample()) # Get a random observation
```

\_\_\_\_OBSERVATION SPACE\_\_\_\_

Observation Space Discrete(16)

Sample observation 9

We see with Observation Space Shape Discrete(16) that the observation is a value representing the **agent's current position as current\_row \* nrows + current\_col (where both the row and col start at 0)**.

For example, the goal position in the 4x4 map can be calculated as follows:  $3 * 4 + 3 = 15$ . The number of possible observations is dependent on the size of the map. **For example, the 4x4 map has 16 possible observations.**

For instance, at this state = 0

In [79]:

```
1 print("\n ____ACTION SPACE____ \n")
2 print("Action Space Shape", env.action_space.n)
3 print("Action Space Sample", env.action_space.sample()) # Take a random action
```

\_\_\_\_ACTION SPACE\_\_\_\_

Action Space Shape 4

Action Space Sample 0

The action space (the set of possible actions the agent can take) is discrete with 4 actions available 🕹:

- 0: GO LEFT
- 1: GO DOWN
- 2: GO RIGHT
- 3: GO UP

Reward function 💰:

- Reach goal: +1
- Reach hole: 0
- Reach frozen: 0

### 3.2.2 Create and Initialize the Q-table

It's time to initialize our Q-table! To know how many rows (states) and columns (actions) to use, we need to know the action and observation space. OpenAI Gym provides us a way to do that: `env.action_space.n` and `env.observation_space.n`

In [80]:

```

1 state_space = env.observation_space.n
2 print("There are ", state_space, " possible states")
3
4 action_space = env.action_space.n
5 print("There are ", action_space, " possible actions")

```

There are 16 possible states

There are 4 possible actions

In [81]:

```

1 # Let's create our Qtable of size (state_space, action_space) and initialized each values at 0
2 def initialize_q_table(state_space, action_space):
3     Qtable = np.zeros((state_space, action_space))
4     return Qtable

```

In [82]:

```
1 Qtable_frozenlake = initialize_q_table(state_space, action_space)
```

### 3.2.3 Define the epsilon-greedy policy

In [83]:

```

1 def epsilon_greedy_policy(Qtable, state, epsilon):
2     # Randomly generate a number between 0 and 1
3     random_int = random.uniform(0, 1)
4     # if random_int > greater than epsilon --> exploitation
5     if random_int > epsilon:
6         # Take the action with the highest value given a state
7         # np.argmax can be useful here
8         action = np.argmax(Qtable[state])
9     # else --> exploration
10    else:
11        action = env.action_space.sample()
12
13    return action

```

### 3.2.4 Define the hyperparameters

The exploration related hyperparamters are some of the most important ones.

- We need to make sure that our agent **explores enough the state space** in order to learn a good value approximation, in order to do that we need to have progressive decay of the epsilon.
- If you decrease too fast epsilon (too high `decay_rate`), **you take the risk that your agent is stuck**, since your agent didn't explore enough the state space and hence can't solve the problem.

In [85]:

```

1 # Training parameters
2 n_training_episodes = 10000 # Total training episodes
3 learning_rate = 0.7 # Learning rate
4
5 # Evaluation parameters
6 n_eval_episodes = 100 # Total number of test episodes
7
8 # Environment parameters
9 env_id = "FrozenLake-v1" # Name of the environment
10 max_steps = 99 # Max steps per episode
11 gamma = 0.95 # Discounting rate
12 eval_seed = [] # The evaluation seed of the environment
13
14 # Exploration parameters
15 max_epsilon = 1.0 # Exploration probability at start
16 min_epsilon = 0.05 # Minimum exploration probability
17 decay_rate = 0.0005 # Exponential decay rate for exploration prob

```

### 3.2.5 Training the model

In [86]:

```

1 def train(n_training_episodes, min_epsilon, max_epsilon, decay_rate, env, max_steps, Qtable):
2     bar = tqdm.tqdm(total=n_training_episodes)
3     for episode in range(n_training_episodes):
4         # Reduce epsilon (because we need less and less exploration)
5         epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.exp(-decay_rate*episode)
6         # Reset the environment
7         state = env.reset()
8         step = 0
9         done = False
10
11        # repeat
12        for step in range(max_steps):
13            # Choose the action At using epsilon greedy policy
14            action = epsilon_greedy_policy(Qtable, state, epsilon)
15
16            # Take action At and observe Rt+1 and St+1
17            # Take the action (a) and observe the outcome state(s') and reward (r)
18            new_state, reward, done, info = env.step(action)
19
20            # Update Q(s,a) := Q(s,a) + lr [R(s,a) + gamma * max Q(s',a') - Q(s,a)]
21            Qtable[state][action] = Qtable[state][action] + learning_rate * (reward + gamma * np.max
22
23            # If done, finish the episode
24            if done:
25                break
26
27            # Our state is the new state
28            state = new_state
29            bar.update()
30    return Qtable

```

In [87]:

```
1 | Qtable_frozenlake = train(n_training_episodes, min_epsilon, max_epsilon, decay_rate, env, max_s
```

100%|██████████| 10000/10000 [00:01<00:00, 5317.84it/s]

### 3.2.6 Trained Q-Learning table

In [88]:

```
1 | Qtable_frozenlake
```

Out[88]:

```
array([[0.73509189, 0.77378094, 0.77378094, 0.73509189],
       [0.73509189, 0.          , 0.81450625, 0.77378094],
       [0.77378094, 0.857375  , 0.77378094, 0.81450625],
       [0.81450625, 0.          , 0.77378094, 0.77378094],
       [0.77378094, 0.81450625, 0.          , 0.73509189],
       [0.          , 0.          , 0.          , 0.          ],
       [0.          , 0.9025    , 0.          , 0.81450625],
       [0.          , 0.          , 0.          , 0.          ],
       [0.81450625, 0.          , 0.857375  , 0.77378094],
       [0.81450625, 0.9025    , 0.9025    , 0.          ],
       [0.857375  , 0.95      , 0.          , 0.857375  ],
       [0.          , 0.          , 0.          , 0.          ],
       [0.          , 0.          , 0.          , 0.          ],
       [0.9025    , 0.95      , 1.          , 0.9025    ],
       [0.          , 0.          , 0.          , 0.          ]])
```

### 3.2.7 Model evaluation

- Normally you should have mean reward of 1.0
- It's relatively easy since the state space is really small (16). What you can try to do is [to replace with the slippery version \(\[https://www.gymlibrary.ml/environments/toy\\\_text/frozen\\\_lake/\]\(https://www.gymlibrary.ml/environments/toy\_text/frozen\_lake/\)\)](https://www.gymlibrary.ml/environments/toy_text/frozen_lake/).

In [89]:

```

1 def evaluate_agent(env, max_steps, n_eval_episodes, Q, seed):
2     """
3         Evaluate the agent for ``n_eval_episodes`` episodes and returns average reward and std of reward
4     :param env: The evaluation environment
5     :param n_eval_episodes: Number of episode to evaluate the agent
6     :param Q: The Q-table
7     :param seed: The evaluation seed array (for taxi-v3)
8     """
9     episode_rewards = []
10    for episode in range(n_eval_episodes):
11        if seed:
12            state = env.reset(seed=seed[episode])
13        else:
14            state = env.reset()
15        step = 0
16        done = False
17        total_rewards_ep = 0
18
19        for step in range(max_steps):
20            # Take the action (index) that have the maximum expected future reward given that state
21            action = np.argmax(Q[state][:])
22            new_state, reward, done, info = env.step(action)
23            total_rewards_ep += reward
24
25            if done:
26                break
27            state = new_state
28        episode_rewards.append(total_rewards_ep)
29    mean_reward = np.mean(episode_rewards)
30    std_reward = np.std(episode_rewards)
31
32    return mean_reward, std_reward

```

In [90]:

```

1 # Evaluate our Agent
2 mean_reward, std_reward = evaluate_agent(env, max_steps, n_eval_episodes, Qtable_frozenlake, eval=True)
3 print(f"Mean_reward={mean_reward:.2f} +/- {std_reward:.2f}")

```

Mean\_reward=1.00 +/- 0.00

### 3.3 Visualizing the results

In [95]:

```

1 def record_video(env, Qtable, out_directory, fps=1):
2     images = []
3     done = False
4
5     state = env.reset(seed=random.randint(0, 500))
6     #state = env.reset()
7     img = env.render(mode='rgb_array')
8     images.append(img)
9     while not done:
10         # Take the action (index) that have the maximum expected future reward given that state
11         action = np.argmax(Qtable[state][:])
12         state, reward, done, info = env.step(action) # We directly put next_state = state for recording
13         img = env.render(mode='rgb_array')
14         images.append(img)
15     imageio.mimsave(out_directory, [np.array(img) for i, img in enumerate(images)], fps=fps)

```

Saving animated file as gif with 1 frame per second

In [96]:

```

1 video_path="replay.gif"
2 video_fps=1

```

In [97]:

```
1 record_video(env, Qtable_frozenlake, video_path, video_fps)
```

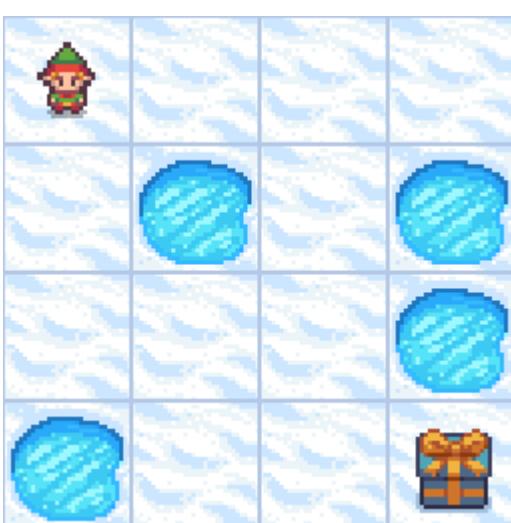
In [98]:

```

1 from IPython.display import Image
2 Image('replay.gif')

```

Out[98]:



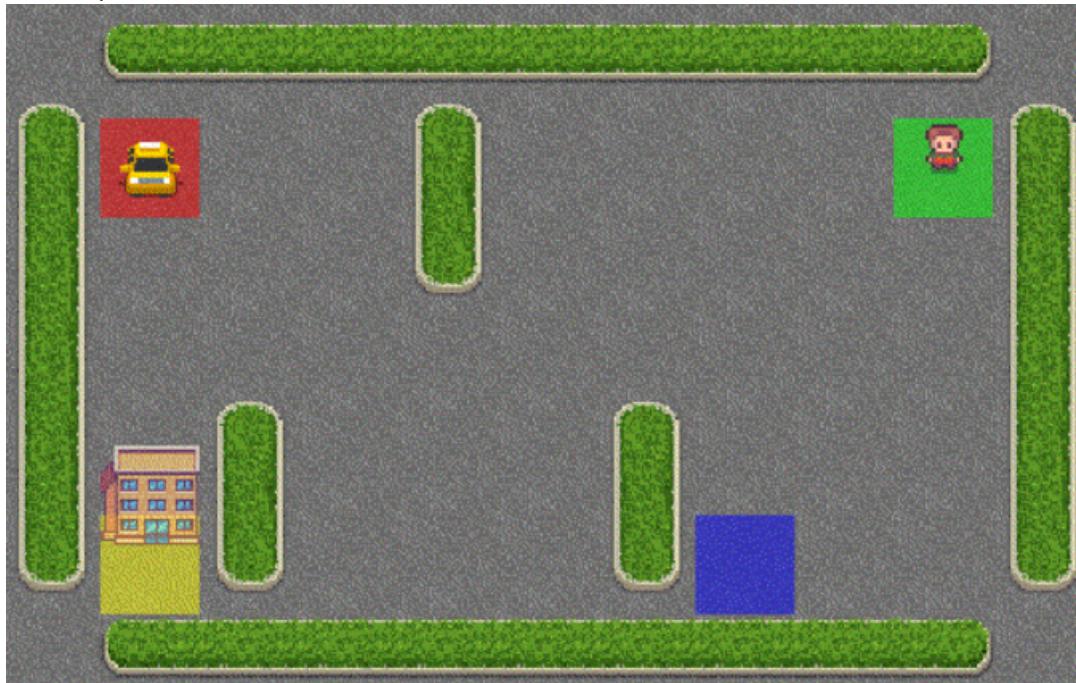
## 4 LAB Assignment

Please finish the **Exercise** and answer **Questions**.

## 4.1 Exercise ( Q-Learning with Taxi-v3 🚕 ) (100 Points)

In this exercise, you should complete the Q-learning algorithm using the Taxi-v3 environment in the gym

In Taxi-v3 🚕, there are four designated locations in the grid world indicated by R(ed), G(reen), Y(ellow), and B(lue). When the episode starts, the taxi starts off at a random square and the passenger is at a random location. The taxi drives to the passenger's location, picks up the passenger, drives to the passenger's destination (another one of the four specified locations), and then drops off the passenger. Once the passenger is dropped off, the episode ends.



## 4.2 Step 0 Import the packages

In [ ]:

```
1 import numpy as np
2 import gym
3 import random
4 import imageio
5 import tqdm
```

Warning: Gym version v0.24.0 has a number of critical issues with `gym.make` such that at the `reset` and `step` functions are called before returning the environment. It is recommended to downgrading to v0.23.1 or upgrading to v0.25.1

## 4.3 Step 1 Create Taxi-v3 🚕 environment

Using the API imported from gym

In [ ]:

```
1 env = gym.make('Taxi-v3')
2 env.render()
```

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**.

In [ ]:

```
1 state_space = env.observation_space.n
2 print("There are ", state_space, " possible states")
```

There are 500 possible states

In [ ]:

```
1 action_space = env.action_space.n
2 print("There are ", action_space, " possible actions")
```

There are 6 possible actions

The action space (the set of possible actions the agent can take) is discrete with **6 actions available** 🕹:

- 0: move south
- 1: move north
- 2: move east
- 3: move west
- 4: pickup passenger
- 5: drop off passenger

Reward function 💰:

- -1 per step unless other reward is triggered.
- +20 delivering passenger.
- -10 executing “pickup” and “drop-off” actions illegally.

## 4.4 Step 2 Create the Q-table and initialize it

You can use the gym api to fetch the dimension of action space and state space

In [ ]:

```
1 action_space = env.action_space.n
2 state_space = env.observation_space.n
3
4 # Please complete this initialization in this line
5 Q_table = None
```

## 4.5 Step 3 Configure the hyperparameters

In [ ]:

```
1 total_episodes = 100000      # 一共玩多少局游戏
2 total_test_episodes = 100     # 测试中一共走几步
3 max_steps = 99               # Max steps per episode 每一局游戏最多走几步
4
5 learning_rate = 0.5          # Learning rate
6 gamma = 0.95                 # Discounting rate
7
8 # Exploration parameters
9 epsilon = 1.0                # Exploration rate
10 max_epsilon = 1.0            # Exploration probability at start
11 min_epsilon = 0.05           # Minimum exploration probability
12 decay_rate = 0.008           # Exponential decay rate for exploration prob
13
14
15 test_seed = [16, 54, 165, 177, 191, 191, 120, 80, 149, 178, 48, 38, 6, 125, 174, 73, 50, 172, 100, 148, 146, 6, 25, 40
16   161, 131, 184, 51, 170, 12, 120, 113, 95, 126, 51, 98, 36, 135, 54, 82, 45, 95, 89, 59, 95, 124, 9, 113, 58, 85, 51, 134,
17   112, 102, 168, 123, 97, 21, 83, 158, 26, 80, 63, 5, 81, 32, 11, 28, 148] # Evaluation seed, this ensures that
18
```



## 4.6 Step 4 Q Learning algorithm

Note: The formula of Q table update(Bellman equation)

Bellman equation

In [ ]:

```

1 bar = tqdm.tqdm(total=total_episodes)
2 sample_rewards = []
3 for episode in range(total_episodes):
4     state= env.reset()
5     step=0
6     done=False
7     sample_reward = 0
8     while True:
9         #TODO: Please complete this action selection in this line via the maximum value
10        action = None
11
12        # TODO:fetech the new state and reward by gym API
13        new_state, reward, done, info = None
14        # Calculate the reward of this episode
15        sample_reward += reward
16
17        # TODO: Update the Q table
18        #  $Q(s, a) := Q(s, a) + lr [R(s, a) + \gamma \max Q(s', a') - Q(s, a)]$ 
19        Q_table[state, action] = None
20
21        # Update the state
22        state = new_state
23
24        #store the episode reward
25        if done == True:
26            sample_rewards.append(sample_reward)
27            break
28        # Reduced exploration probability (due to decreasing uncertainty)
29        epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.exp(-decay_rate*episode)
30        # print the average reward over 1000 episodes
31        if episode%1000 == 0:
32            mean_reward = np.mean(sample_rewards)
33            sample_rewards = []
34            #print(str(episode)+": average reward:" + str(mean_reward))
35            bar.set_description(str(episode)+": average reward:" + str(mean_reward))
36            bar.update()

```

## 4.7 Step 5 Model evaluation

In [ ]:

```
1 fps=5
2 bar = tqdm.tqdm(total=total_test_episodes)
3 env.reset()
4 rewards=[]
5 images = []
6 for episode in range(total_test_episodes):
7     state = env.reset(seed=test_seed[episode])
8     step = 0
9     done = False
10    total_rewards = 0
11
12    for step in range(max_steps):
13        img = env.render(mode='rgb_array')
14        images.append(img)
15        #TODO:action selection
16        action = None
17        #TODO:fetech the new state and reward by gym API
18        new_state, reward, done, info = None
19
20        total_rewards += reward
21        if done:
22            rewards.append(total_rewards)
23            break
24        state = new_state
25
26 env.close()
27 mean_reward = np.mean(rewards)
28 std_reward = np.std(rewards)
29 print(f"Mean_reward={mean_reward:.2f} +/- {std_reward:.2f}")
30 imageio.mimsave('taxi-v3.gif', [np.array(img) for i, img in enumerate(images)], fps=fps)
31
```

## 4.8 Step 6 Visualizing the results

In [ ]:

```
1 from IPython.display import Image  
2 Image('taxi-v3.gif')
```

