

Q-learning for beginners

Train an AI to solve the Frozen Lake environment



Maxime Labonne  · Follow

Published in Towards Data Science

23 min read · Mar 7, 2022

 Listen

 Share

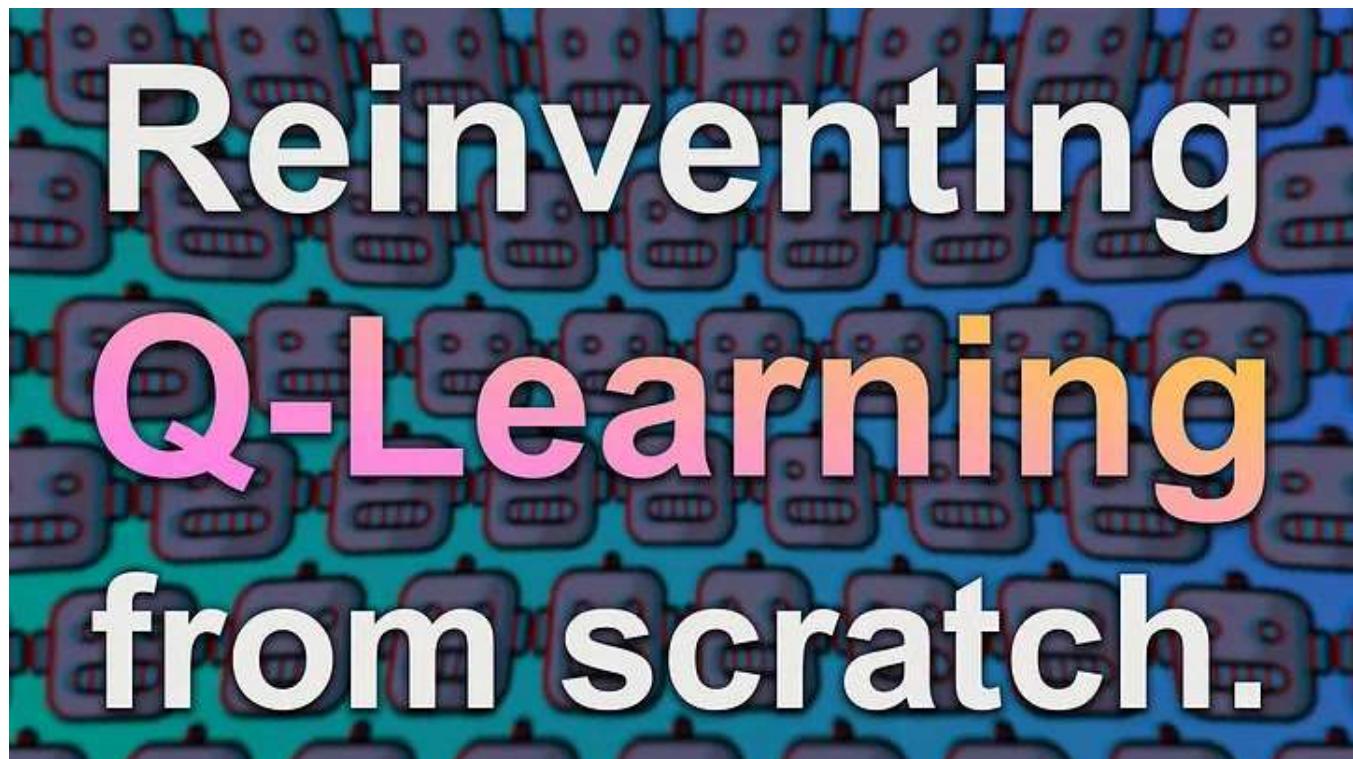


Image by author

The goal of this article is to teach an AI how to solve the  **Frozen Lake** environment using reinforcement learning. We're going to start from scratch and try to recreate the Q-learning algorithm by ourselves. We'll not just understand how it works, but more importantly, why it was designed that way.

By the end of this article, you'll master the Q-learning algorithm and be able to apply it to other environments. It's a cool mini-project that gives a better insight into how

reinforcement learning works and can hopefully inspire ideas for original and creative applications.

Let's start by installing the Frozen Lake environment and importing the necessary libraries: `gym` for the game, `random` to generate random numbers, and `numpy` to do some math.

```
!pip install -q gym
!pip install -q matplotlib
import gym
import random
import numpy as np
```

✳️ I. Frozen Lake

Now, let's talk about the game we're going to be solving in this tutorial. Frozen Lake is a simple environment composed of tiles, where the AI has to move from an initial tile to a goal.

- Tiles can be a safe frozen lake  , or a hole  that gets you stuck forever.
- The AI, or agent, has 4 possible actions: go  LEFT,  DOWN,  RIGHT, or  UP.
- The agent must learn to avoid holes in order to reach the goal in a minimal number of actions.

By default, the environment is always in the same configuration. In the environment's code, each tile is represented by a letter as follows:

S F F F	(S: starting point, safe)
F H F H	(F: frozen surface, safe)
F F F H	(H: hole, stuck forever)
H F F G	(G: goal, safe)

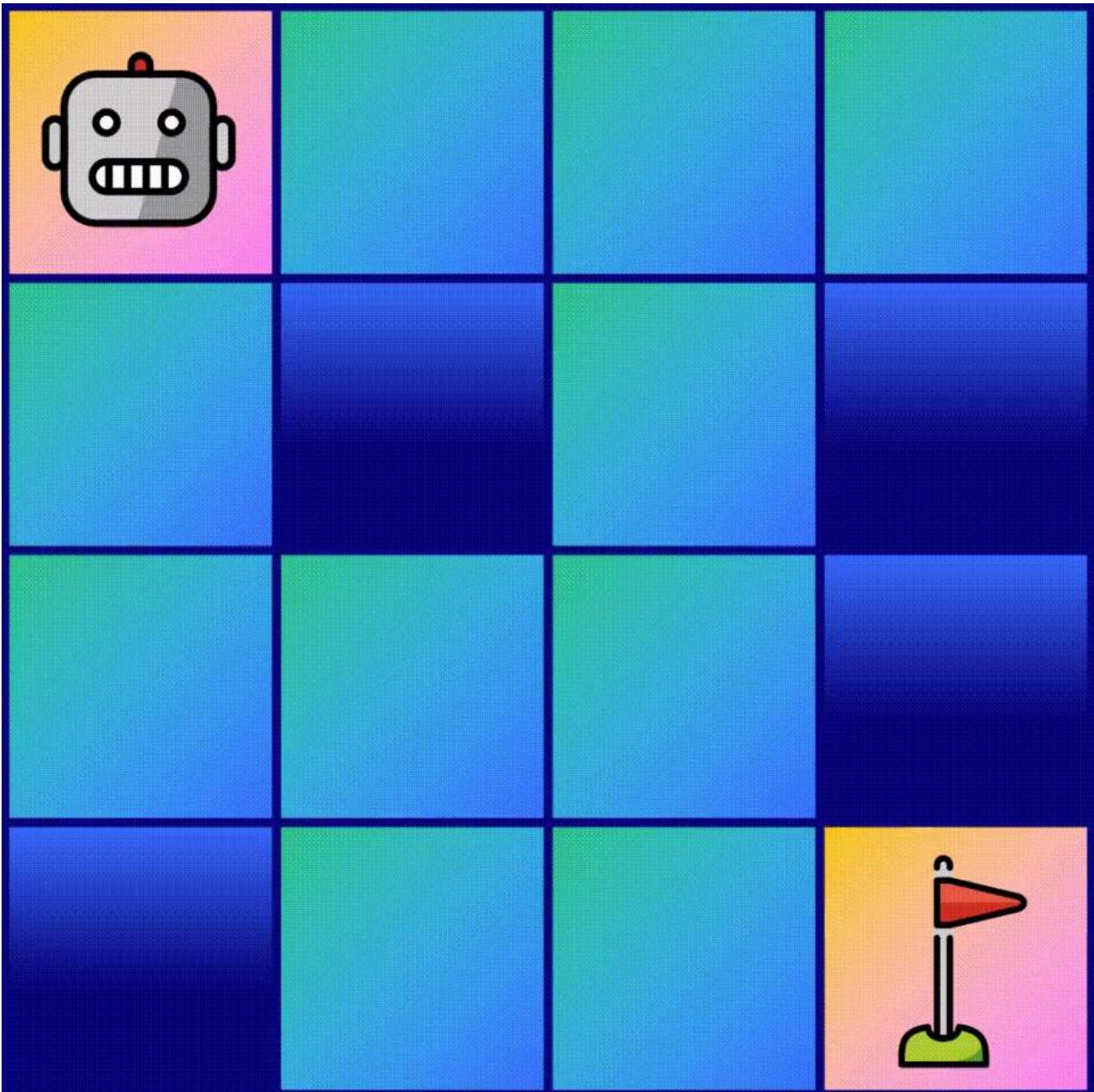


Image by author

We can try to manually solve the example above to understand the game. Let's see if the following sequence of actions is a correct solution: **RIGHT** → **RIGHT** → **RIGHT** → **DOWN** → **DOWN** → **DOWN**. Our agent starts on tile **S**, so we move right on a frozen surface ✓, then again ✓, then once more ✓, then we go down and find a hole ✗.

Actually, it's really easy to find several correct solutions: **RIGHT** → **RIGHT** → **DOWN** → **DOWN** → **RIGHT** is an obvious one. But we could make a sequence of actions that loops around a hole 10 times before reaching the goal. This sequence is valid, but it doesn't meet our final requirement: **the agent needs to meet the goal in a minimum number of actions**. In this example, the minimum number of actions to

complete the game is **6**. We need to remember this fact to check if our agent really masters Frozen Lake or not.

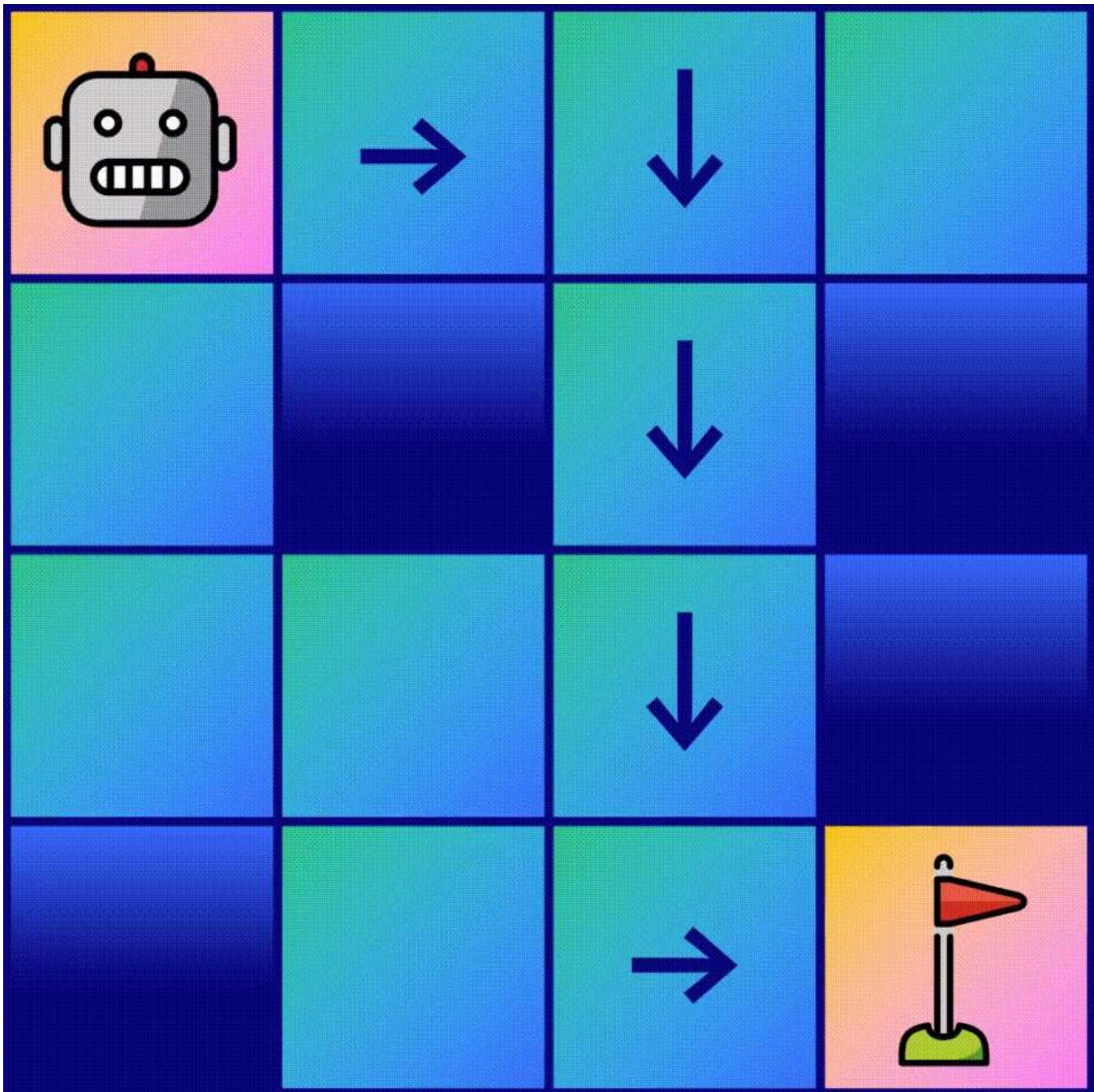


Image by author

Let's initialize the environment thanks to the `gym` library. There are two versions of the game: one with slippery ice, where selected actions have a random chance of being disregarded by the agent; and a non-slippery one, where actions cannot be ignored. We'll use the non-slippery one to begin with because it's easier to understand.

```
environment = gym.make("FrozenLake-v1", is_slippery=False)
environment.reset()
environment.render()
```

 FFF

FHFH

FFFFH

HFFG

We can see that the game that was created has the exact same configuration as in our example: it is the same puzzle. The position of our agent is indicated by a red rectangle. Solving this puzzle can be done with a simple script and if...else conditions, which would actually be useful to compare our AI to a simpler approach. However, we want to try a more exciting solution: **reinforcement learning**.

🏁 II. Q-table

In Frozen Lake, there are 16 tiles, which means our agent can be found in 16 different positions, called **states**. For each state, there are 4 possible actions: go  LEFT,  DOWN,  RIGHT, and  UP. Learning how to play Frozen Lake is like learning which action you should choose in every state. To know which action is the best in a given state, we would like to assign a quality value to our actions. We have 16 states and 4 actions, so want to calculate $16 \times 4 = 64$ values.

A nice way of representing it is using a table, known as a Q-table, where rows list every state s and columns list every action a . In this Q-table, each cell contains a value $Q(s, a)$, which is the value (quality) of the action a in the state s (1 if it's the best action possible, 0 if it's really bad). When our agent is in a particular state s , it just has to check this table to see which action has the highest value. Taking the action with the highest value makes sense but we'll see later that we can design something even better...

S	 LEFT	 DOWN	 RIGHT	 UP
0	$Q(0, \leftarrow)$	$Q(0, \downarrow)$	$Q(0, \rightarrow)$	$Q(0, \uparrow)$
1	$Q(1, \leftarrow)$	$Q(1, \downarrow)$	$Q(1, \rightarrow)$	$Q(1, \uparrow)$
2	$Q(2, \leftarrow)$	$Q(2, \downarrow)$	$Q(2, \rightarrow)$	$Q(2, \uparrow)$
...

14	Q(14, ⬅)	Q(14, ⬇)	Q(14, ⏴)	Q(14, ⏵)
G	Q(15, ⬅)	Q(15, ⬇)	Q(15, ⏴)	Q(15, ⏵)

Let's create our Q-table and fill it with zeros since we still have no idea of the value of each action in each state.

```
# Our table has the following dimensions:  
# (rows x columns) = (states x actions) = (16 x 4)  
qtable = np.zeros((16, 4))  
  
# Alternatively, the gym library can also directly give us the number of states and actions using  
# "env.observation_space.n" and "env.action_space.n"  
nb_states = environment.observation_space.n # = 16  
nb_actions = environment.action_space.n # = 4  
qtable = np.zeros((nb_states, nb_actions))  
  
# Let's see how it looks  
print('Q-table =')  
print(qtable)
```

```
Q-table =  
[[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. 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. 0. 0.]  
 [0. 0. 0. 0.]  
 [0. 0. 0. 0.]  
 [0. 0. 0. 0.]]
```

Great! We have our Q-table with 16 rows (our 16 states) and 4 columns (our 4 actions) as expected. Let's try to see what we can do next: every value is set to zero, so we have no information at all. Let's say that the agent takes a random action: ⬅ LEFT, ⬇ DOWN, ⏴ RIGHT, or ⏵ UP.

We can use the `random` library with the `choice` method to randomly choose an action.

```
random.choice(["LEFT", "DOWN", "RIGHT", "UP"])
```

'LEFT'

Wait, actually the agent is currently on the initial state **S**, which means only two actions are possible: **RIGHT** and **DOWN**. The agent can also take the actions **UP** and **LEFT**, but it won't move: its state doesn't change. Therefore, we do not put any constraint on what actions are possible: the agent will naturally understand that some of them don't do anything.

We can keep using `random.choice()`, but the `gym` library already implements a method to randomly choose an action. It might save us some hassle later, so let's try it.

```
environment.action_space.sample()
```

0

Oops... this time it's a **number**. We could read `gym`'s [documentation](#) but it is quite scarce, unfortunately. No worries though, [we can check the source code on GitHub](#) to understand what these numbers mean. It's actually super straightforward:

- LEFT = 0
- DOWN = 1
- RIGHT = 2
- UP = 3

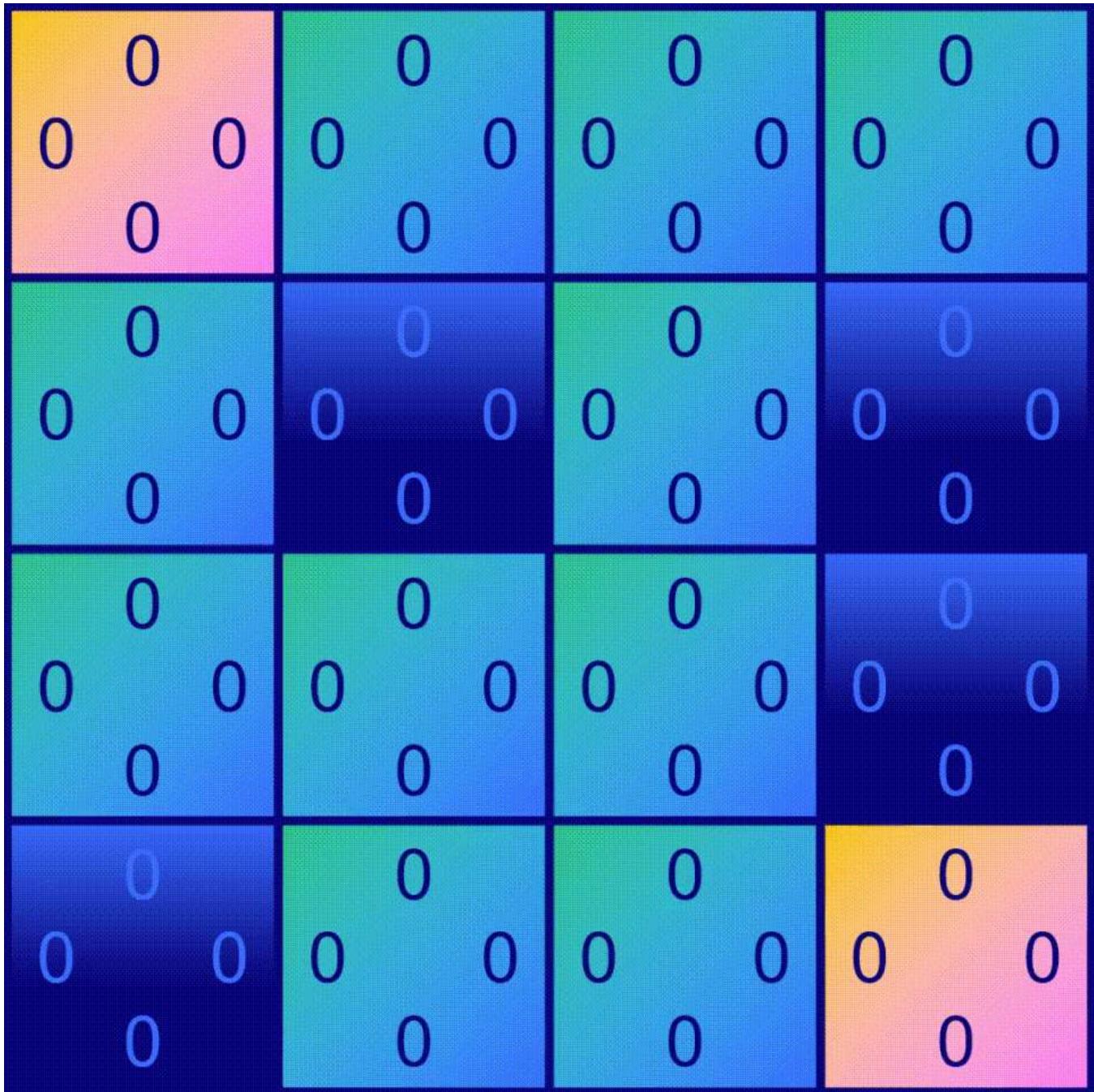


Image by author

Okay, now that we understand how `gym` connects numbers to directions, let's try to use it to move our agent to the right ➤. This time, it can be performed using the `step(action)` method. We can try to directly provide it the number 2, corresponding to the direction we chose (right), and check if the agent moved.

```
environment.step(2)  
environment.render()
```

(Right)
S █ FF

FHFH
FFFH
HFFG

Huzzah! The red square moved from the initial state S to the right: our prediction was correct. And that's all we need to know in order to interact with the environment:

1. How to **randomly choose** an action using `action_space.sample()` ;
2. How to **implement** this action and move our agent in the desired direction with `step(action)` .

To be completely exhaustive, we can add:

1. How to **display** the current map to see what we're doing with `render()` ;
2. How to **restart** the game when the agent falls into a hole or reaches the goal G with `reset()` .

Now that we understand how to interact with our `gym` environment, let's go back to our algorithm. In reinforcement learning, agents are rewarded by the environment when they accomplish a predefined goal. In Frozen Lake, the agent is only rewarded when it reaches the state G (see [the source code](#)). We cannot control this reward, it is set in the environment: it's 1 when the agent reaches G, and 0 otherwise.

Let's print it every time we implement an action. The reward is given by the method `step(action)` .

```
action = environment.action_space.sample()

# 2. Implement this action and move the agent in the desired direction
new_state, reward, done, info = environment.step(action)

# Display the results (reward and map)
environment.render()
print(f'Reward = {reward}')
```

(Left)
 FFF

FHFH
FFFH
HFFG
Reward = 0.0

The reward is indeed 0... 🤯 wow, I guess we're in a pickle, because only one state can give us a positive reward in the entire game. How are we supposed to take the right directions at the very beginning when the only validation we have is at the very end? If we ever want to see a reward of 1, we'd need to be lucky enough to find the correct sequence of actions by chance. Unfortunately, that's exactly how it works... the Q-table will remain **filled with zeros** until the agent randomly reaches the goal G.

The problem would be much simpler if we could have intermediate, smaller rewards to guide our path towards the goal G. Alas, this is actually one of the main issues of reinforcement learning: this phenomenon, called **sparse rewards**, makes agents very difficult to train on problems where the only reward is at the end of a long sequence of actions. Different techniques were proposed to mitigate this issue, but we'll talk about it another time.

III. Q-learning

Let's go back to our problem. Okay, we need to be lucky enough to find the goal G by accident. But once it's done, how to backpropagate the information to the initial state? The Q-learning algorithm offers a clever solution to this issue. We need to **update the value** of our state-action pairs (each cell in the Q-table) considering 1/ the reward for reaching the next state, and 2/ the highest possible value in the next state.

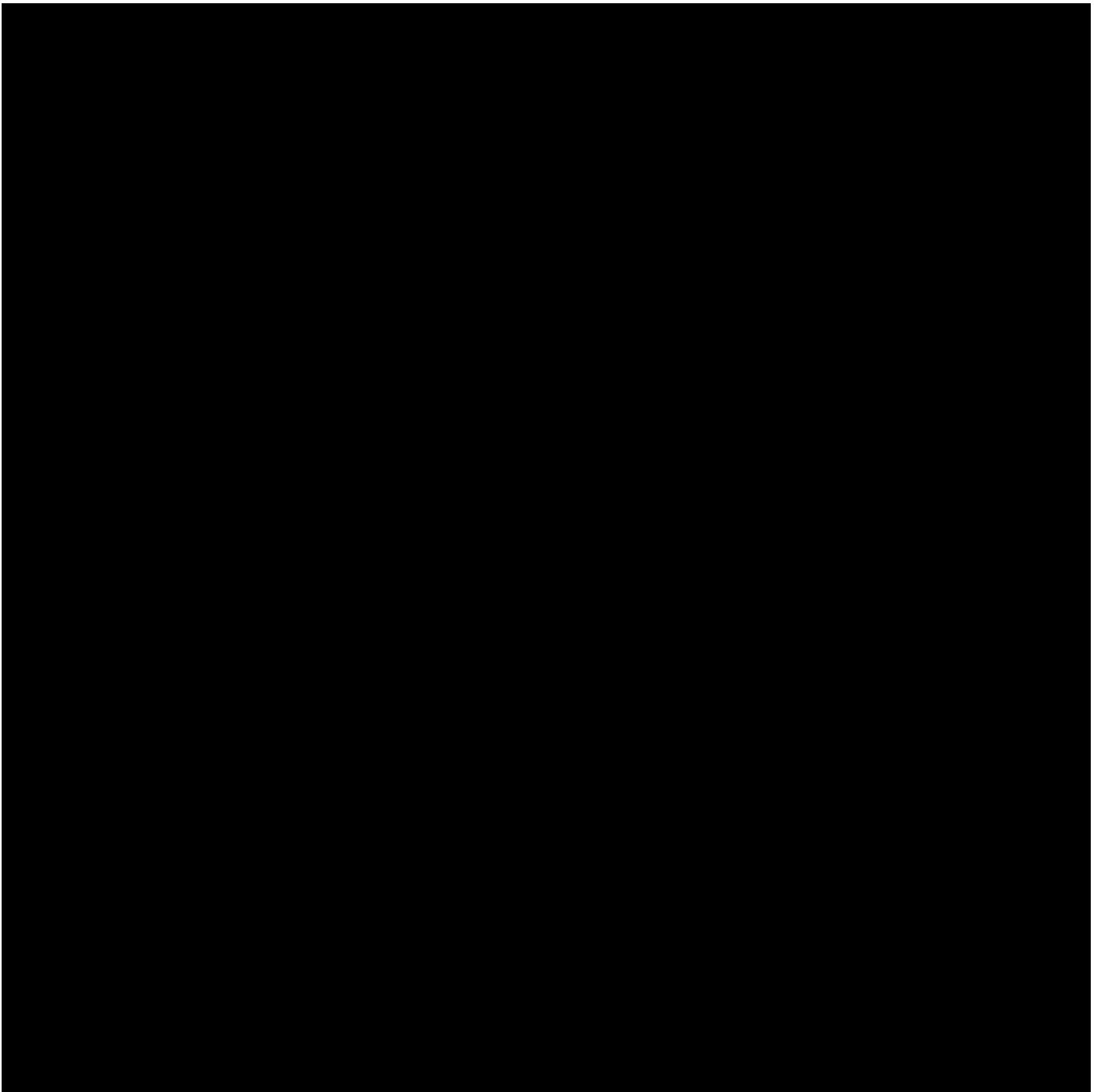


Image by author

We know we get a reward of 1 when we move to **G**. As we just said, the value of the state next to **G** (let's call it **G-1**) with the relevant action to reach **G** is increased thanks to the reward. Okay good, end of the episode: the agent won and we restarted the game. Now, the next time the agent is in a state next to **G-1**, it will increase the value of this state (let's call it **G-2**) with the relevant action to reach **G-1**. The next time the agent is in a state next to **G-2**, it will do the same. Rinse and repeat, until the update reaches the initial state **S**.

Let's try to find the update formula to backpropagate the values from **G** to **S**. Remember: values denote the quality of an action in a specific state (0 if it's terrible, 1 if it's the best action possible in this state). We try to update the value of the action

a_t (for example, $a_t = 0$ if the action is left) in the state s_t (for example, $s_t = 0$ when the agent is in the initial state S). This value is just a cell in our Q-table, corresponding to the **row number s_t** and the **column number a_t** : this value is formally called $Q(s_t, a_t)$.

As we said previously, we need to update it using 1/ **the reward for the next state** (formally noted r_t), and 2/ **the maximum possible value in the next state** ($\max_a Q(s_{t+1}, a)$). Therefore, the update formula must look like:

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

The new value is the current one + the reward + the highest value in the next state. We can manually try our formula to check if it looks correct: let's pretend our agent is **in the state G-1 next to the goal G for the first time**. We can update the value corresponding to the winning action in this state **G-1** with:

$$Q_{new}(G - 1, a_t) = Q(G - 1, a_t) + r_t + \max_a Q(G, a)$$

where $Q(G-1, a_t) = 0$ and $\max_a Q(G, a) = 0$ because the Q-table is empty, and $r_t = 1$ because we get the only reward in this environment. We obtain $Q\{new\}(G-1, a_t) = 1$. The next time the agent is in a state next to this one (**G-2**), we update it too using the formula, and get the same result: $Q\{new\}(G-2, a_t) = 1$. In the end, we **backpropagate ones in the Q-table from G to S**. Okay it works, but the result is binary: either it's the wrong state-action pair or the best one. We would like more nuance...

Actually, we almost found the **true Q-learning update formula** with common sense. The nuance we're looking for adds two parameters:

- α is the  **learning rate** (between 0 and 1), which is how much we should change the original $Q(s_t, a_t)$ value. If $\alpha = 0$, the value never changes, but if $\alpha = 1$, the value changes extremely fast. In our attempt, we didn't limit the learning rate so $\alpha = 1$. But this is too fast in reality: the reward and the maximum value in the next state quickly overpower the current value. We need to find a balance between the importance of past and new knowledge.
- γ is the  **discount factor** (between 0 and 1), which determines how much the agent cares about future rewards compared to immediate ones (as the saying goes, "a bird in the hand is worth two in the bush"). If $\gamma = 0$, the agent only focuses on immediate rewards, but if $\gamma = 1$, any potential future reward has the

same value as current ones. In  Frozen Lake, we want a high discount factor since there's only one possible reward at the very end of the game.

With the real Q-learning algorithm, the new value is calculated as follows:

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

Okay, let's try this new formula before implementing it. Once again, we can pretend that our agent is **next to the goal G for the first time**. We can update the state-action pair to win the game using our formula: $Q\{new\}(G-1, a_t) = 0 + \alpha \cdot (1 + \gamma \cdot 0 - 0)$. We can assign arbitrary values to α and γ to calculate the result. With $\alpha = 0.5$ and $\gamma = 0.9$, we get $Q\{new\}(G-1, a_t) = 0 + 0.5 \cdot (1 + 0.9 \cdot 0 - 0) = 0.5$. The second time the agent is in this state, we would get: $Q\{new\}(G-1, a_t) = 0.5 + 0.5 \cdot (1 + 0.9 \cdot 0 - 0.5) = 0.75$, then 0.875, 0.9375, 0.96875, etc.

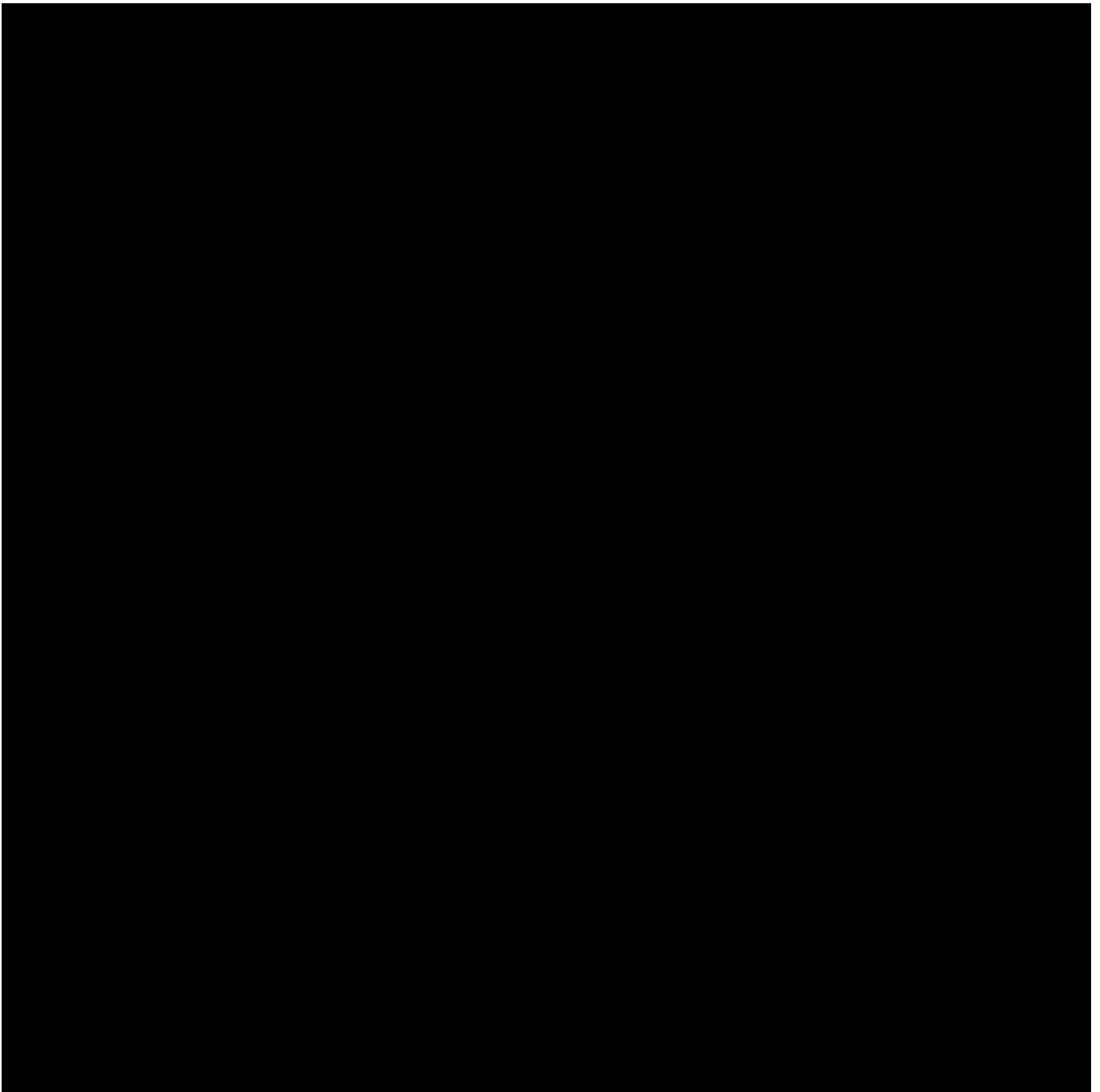


Image by author

So training our agent in code means:

- 1. Choosing a random action** (using `action_space.sample()`) if the values in the current state are just zeros. Otherwise, we take the action with the highest value in the current state with the function `np.argmax()`;
- 2. Implementing this action** by moving in the desired direction with `step(action)`;
- 3. Updating the value** of the original state with the action we took, using information about the new state and the reward given by `step(action)`;

We keep repeating these 3 steps until the agent gets stuck in a hole or reaches the goal G. When it happens, we just restart the environment with `reset()` and start a new episode until we hit 1,000 episodes. Additionally, we can plot the outcome of each run (failure if it didn't reach the goal, success otherwise) to observe the progress of our agent.

```

import matplotlib.pyplot as plt
plt.rcParams['figure.dpi'] = 300
plt.rcParams.update({'font.size': 17})

# We re-initialize the Q-table
qtable = np.zeros((environment.observation_space.n, environment.action_space.n))

# Hyperparameters
episodes = 1000          # Total number of episodes
alpha = 0.5                # Learning rate
gamma = 0.9                # Discount factor

# List of outcomes to plot
outcomes = []

print('Q-table before training:')
print(qtable)

# Training
for _ in range(episodes):
    state = environment.reset()
    done = False

    # By default, we consider our outcome to be a failure
    outcomes.append("Failure")

    # Until the agent gets stuck in a hole or reaches the goal, keep training i
    while not done:
        # Choose the action with the highest value in the current state
        if np.max(qtable[state]) > 0:
            action = np.argmax(qtable[state])

        # If there's no best action (only zeros), take a random one
        else:
            action = environment.action_space.sample()

        # Implement this action and move the agent in the desired direction
        new_state, reward, done, info = environment.step(action)

        # Update Q(s,a)
        qtable[state, action] = qtable[state, action] + \
                               alpha * (reward + gamma * np.max(qtable[new_st

```

```

# Update our current state
state = new_state

# If we have a reward, it means that our outcome is a success
if reward:
    outcomes[-1] = "Success"

print()
print('=====')
print('Q-table after training:')
print(qtable)

# Plot outcomes
plt.figure(figsize=(12, 5))
plt.xlabel("Run number")
plt.ylabel("Outcome")
ax = plt.gca()
ax.set_facecolor('#efeeea')
plt.bar(range(len(outcomes)), outcomes, color="#0A047A", width=1.0)
plt.show()

```

Q-table before training:

```

[[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. 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. 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. 0.]]
```

=====

Q-table after training:

```

[[0.          0.          0.59049   0.          ]
 [0.          0.          0.6561     0.          ]
 [0.          0.729       0.          0.          ]
 [0.          0.          0.          0.          ]
 [0.          0.02050313 0.          0.          ]
 [0.          0.          0.          0.          ]
 [0.          0.81        0.          0.          ]
 [0.          0.          0.          0.          ]
 [0.          0.          0.17085938 0.          ]
 [0.          0.          0.49359375 0.          ]]
```

```
[0.          0.9       0.          0.          ]
[0.          0.         0.          0.          ]
[0.          0.         0.          0.          ]
[0.          0.         0.          0.          ]
[0.          0.         1.          0.          ]
[0.          0.         0.          0.          ]]
```

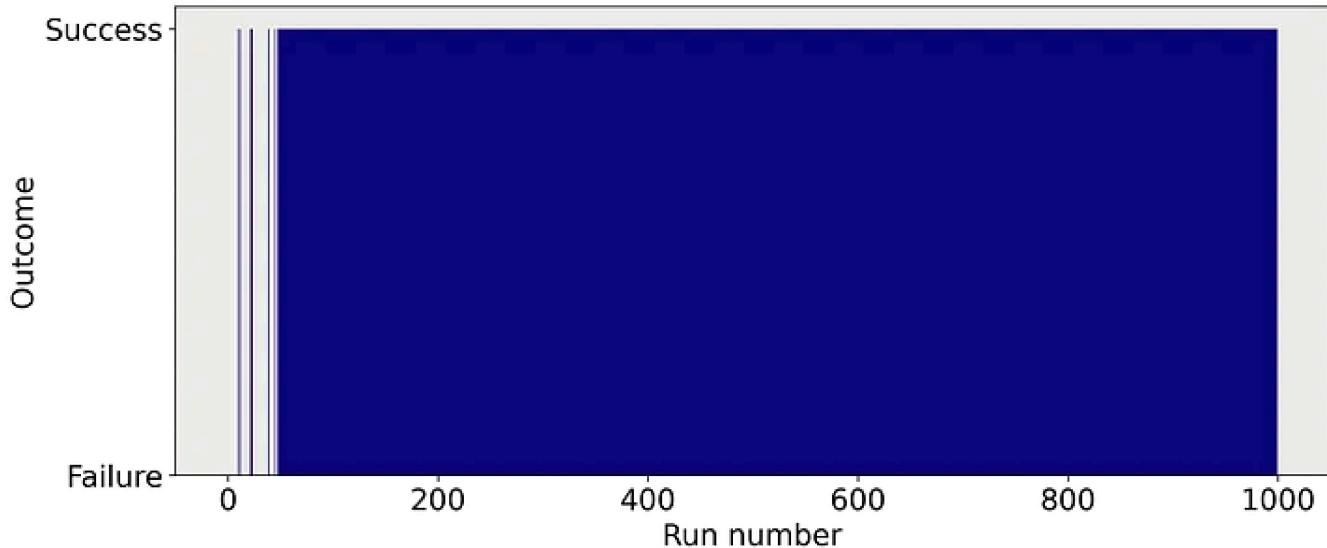


Image by author

The agent is trained! Each blue bar on the figure corresponds to a win, so we can see that the agent had a hard time finding the goal at the beginning of the training. But once it found it several times in a row, it began to **consistently win**. 🎉 The trained Q-table is also very interesting: these values indicate the unique sequence of actions the agent learned to reach the goal.

Now let's see how it performs by evaluating it on 100 episodes. We consider that the training is over, so we don't need to update the Q-table anymore. To see how the agent performs, we can calculate the percentage of times it managed to reach the goal (success rate).

```
episodes = 100
nb_success = 0

# Evaluation
for _ in range(100):
    state = environment.reset()
    done = False

    # Until the agent gets stuck or reaches the goal, keep training it
    while not done:
```

```

# Choose the action with the highest value in the current state
if np.max(qtable[state]) > 0:
    action = np.argmax(qtable[state])

# If there's no best action (only zeros), take a random one
else:
    action = environment.action_space.sample()

# Implement this action and move the agent in the desired direction
new_state, reward, done, info = environment.step(action)

# Update our current state
state = new_state

# When we get a reward, it means we solved the game
nb_success += reward

# Let's check our success rate!
print (f"Success rate = {nb_success/episodes*100}%")

```

Success rate = 100.0%

Not only our agent has been trained, but it has managed to hit a **100% success rate**. Great job everyone, the non-slippery Frozen Lake is solved!

We can even visualize the agent moving on the map by executing the code below and printing the sequence of actions it took to check if it's the best one.

```

from IPython.display import clear_output
import time

state = environment.reset()
done = False
sequence = []

while not done:
    # Choose the action with the highest value in the current state
    if np.max(qtable[state]) > 0:
        action = np.argmax(qtable[state])

    # If there's no best action (only zeros), take a random one
    else:
        action = environment.action_space.sample()

    # Add the action to the sequence
    sequence.append(action)

    # Move the agent and get the next state, reward, etc
    state, reward, done, info = environment.step(action)

```

```

sequence.append(action)

# Implement this action and move the agent in the desired direction
new_state, reward, done, info = environment.step(action)

# Update our current state
state = new_state

# Update the render
clear_output(wait=True)
environment.render()
time.sleep(1)

print(f"Sequence = {sequence}")

```

(Right)
SFFF
FHFH
FFFF
HFF 
Sequence = [2, 2, 1, 1, 1, 2]

The agent can learn several correct sequences of actions: [2, 2, 1, 1, 1, 2], [1, 1, 2, 2, 1, 2], etc. The good thing is there are **only 6 actions in our sequence**, which was the **minimum possible number of actions we counted**: it means that our agent learned to solve the game in an optimal way. In the case of [2, 2, 1, 1, 1, 2], which corresponds to RIGHT → RIGHT → DOWN → DOWN → DOWN → RIGHT, it's exactly the sequence we predicted at the very beginning of the article. 

IV. Epsilon-Greedy algorithm

Despite this success, there's something that bothers me with our previous approach: the agent always chooses the action with the highest value. So whenever a state-action pair starts having a **non-zero value**, the agent will always choose it. The other actions will never be taken, which means we'll never update their value... But what if one of these actions was better than the one the agent always takes? Shouldn't we encourage the agent to try new things from time to time and see if it can improve?

In other words, we want to allow our agent to either:

- Take the action with the highest value (exploitation);

- Choose a random action to try to find even better ones (exploration).

A tradeoff between these two behaviors is important: if the agent only focuses on **exploitation**, it cannot try new solutions and thus doesn't learn anymore. On the other hand, if the agent only takes random actions, the training is pointless since it doesn't use the Q-table. So we want to change this parameter over time: at the beginning of the training, we want to explore the environment as much as possible. But **exploration** becomes less and less interesting, as the agent already knows every possible state-action pair. This parameter represents the amount of randomness in the action selection.

This technique is commonly called the **epsilon-greedy algorithm**, where epsilon is our parameter. It is a simple but extremely efficient method to find a good tradeoff. Every time the agent has to take an action, it has a probability ϵ of choosing a random one, and a probability $1-\epsilon$ of choosing the one with the highest value. We can decrease the value of epsilon at the end of each episode by a fixed amount (**linear decay**), or based on the current value of epsilon (**exponential decay**).

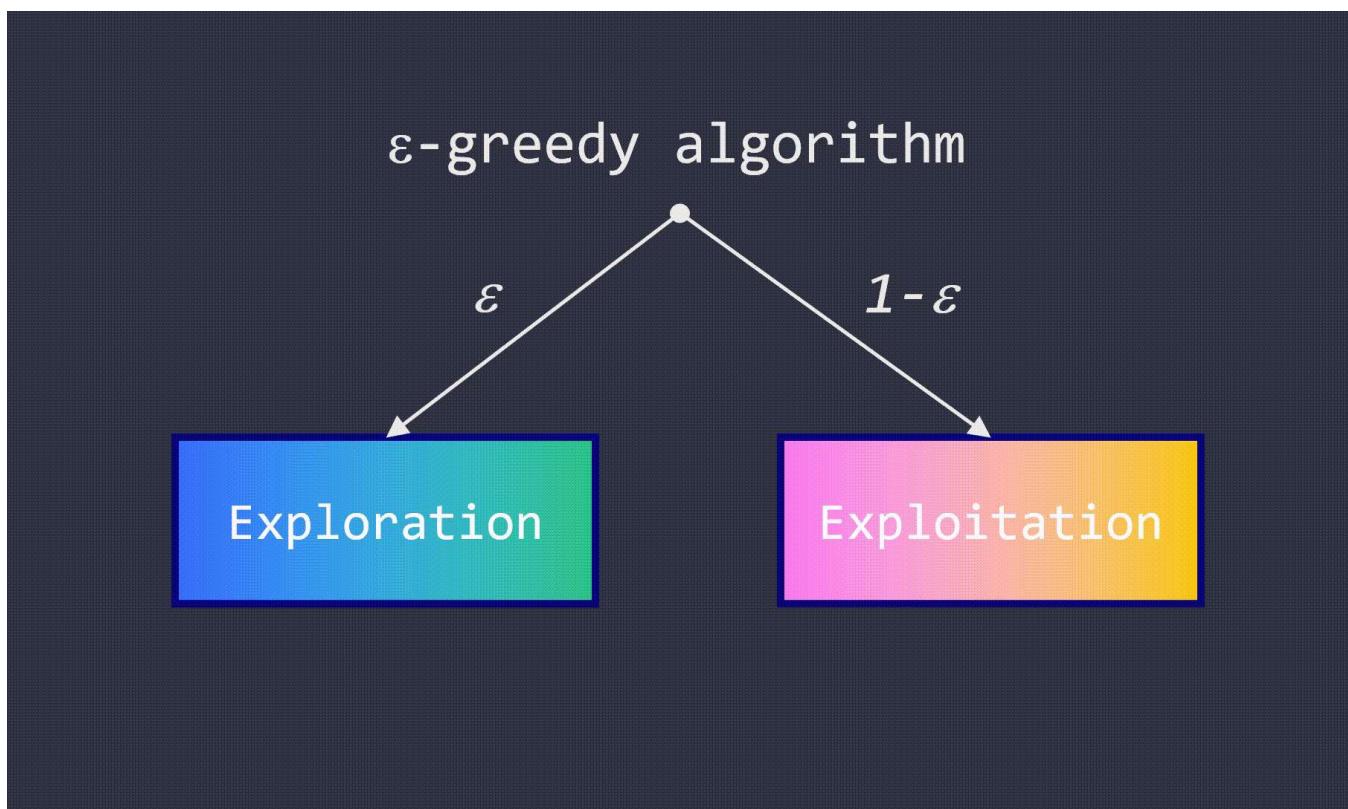


Image by author

Let's implement a **linear decay**. Beforehand, I'd like to see how the curve looks like with arbitrary parameters. We'll start with $\epsilon = 1$ to be in full exploration mode, and decrease this value by 0.001 after each episode.

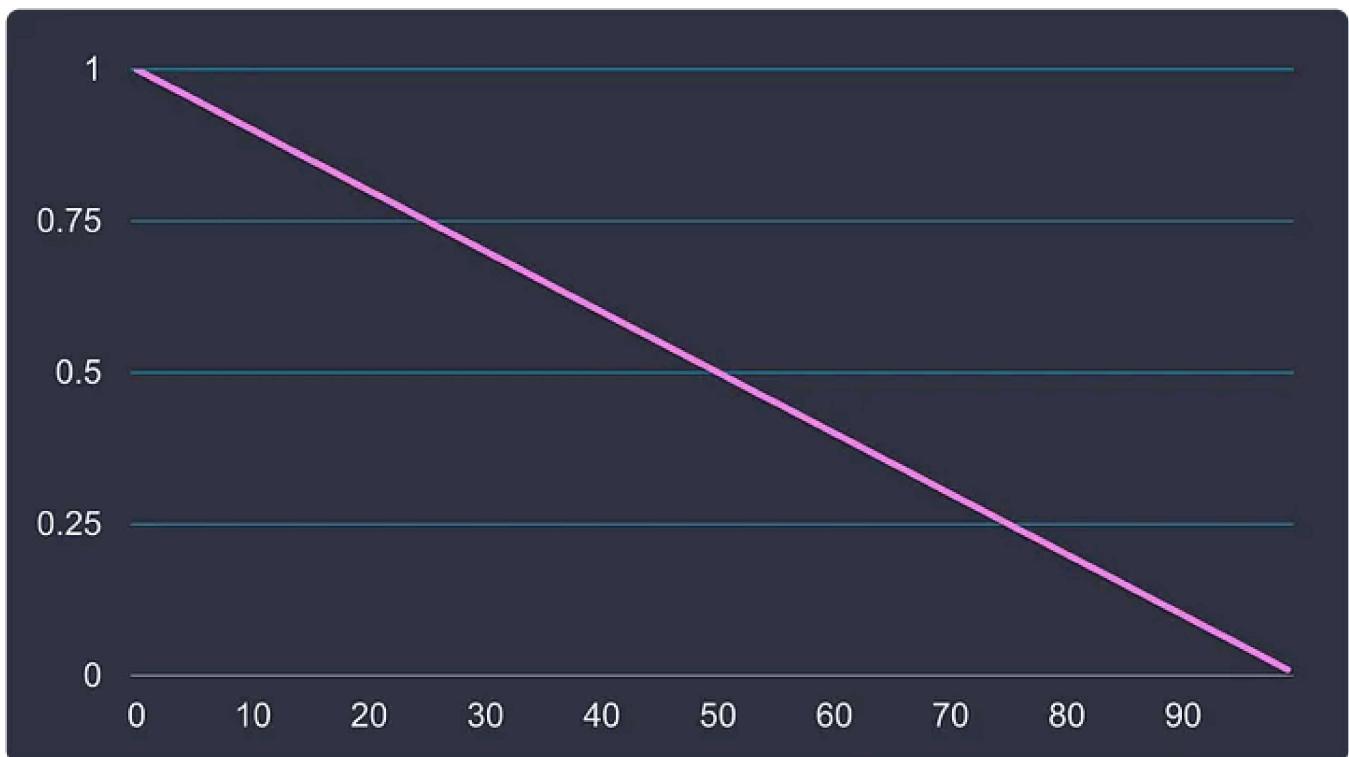


Image by author

Okay now that we have a sound understanding of it, we can implement it for real and see how it changes the agent's behavior.

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

# Hyperparameters
episodes = 1000          # Total number of episodes
alpha = 0.5                # Learning rate
gamma = 0.9                # Discount factor
epsilon = 1.0              # Amount of randomness in the action selection
epsilon_decay = 0.001        # Fixed amount to decrease

# List of outcomes to plot
outcomes = []

print('Q-table before training:')
print(qtable)

# Training
for _ in range(episodes):
    state = environment.reset()
    done = False

    # By default, we consider our outcome to be a failure
    outcomes.append("Failure")

    # Until the agent gets stuck in a hole or reaches the goal, keep training i
    while not done:
```

```

# Generate a random number between 0 and 1
rnd = np.random.random()

# If random number < epsilon, take a random action
if rnd < epsilon:
    action = environment.action_space.sample()
# Else, take the action with the highest value in the current state
else:
    action = np.argmax(qtable[state])

# Implement this action and move the agent in the desired direction
new_state, reward, done, info = environment.step(action)

# Update Q(s,a)
qtable[state, action] = qtable[state, action] + \
                        alpha * (reward + gamma * np.max(qtable[new_st

# Update our current state
state = new_state

# If we have a reward, it means that our outcome is a success
if reward:
    outcomes[-1] = "Success"

# Update epsilon
epsilon = max(epsilon - epsilon_decay, 0)

print()
print('=====')
print('Q-table after training:')
print(qtable)

# Plot outcomes
plt.figure(figsize=(12, 5))
plt.xlabel("Run number")
plt.ylabel("Outcome")
ax = plt.gca()
ax.set_facecolor('#efeeea')
plt.bar(range(len(outcomes)), outcomes, color="#0A047A", width=1.0)
plt.show()

```

Q-table before training:

```

[[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. 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. 0. 0.]  
[0. 0. 0. 0.]  
[0. 0. 0. 0.]  
[0. 0. 0. 0.]  
[0. 0. 0. 0.]
```

=====

Q-table after training:

```
[[0.531441 0.59049 0.59049 0.531441 ]  
 [0.531441 0. 0.6561 0.56396466]  
 [0.58333574 0.729 0.56935151 0.65055117]  
 [0.65308668 0. 0.33420534 0.25491326]  
 [0.59049 0.6561 0. 0.531441 ]  
 [0. 0. 0. 0.]  
 [0. 0.81 0. 0.65519631]  
 [0. 0. 0. 0.]  
 [0.6561 0. 0.729 0.59049 ]  
 [0.6561 0.81 0.81 0.]  
 [0.72899868 0.9 0. 0.72711067]  
 [0. 0. 0. 0.]  
 [0. 0. 0. 0.]  
 [0. 0.81 0.9 0.729 ]  
 [0.81 0.9 1. 0.81 ]  
 [0. 0. 0. 0.] ]]
```

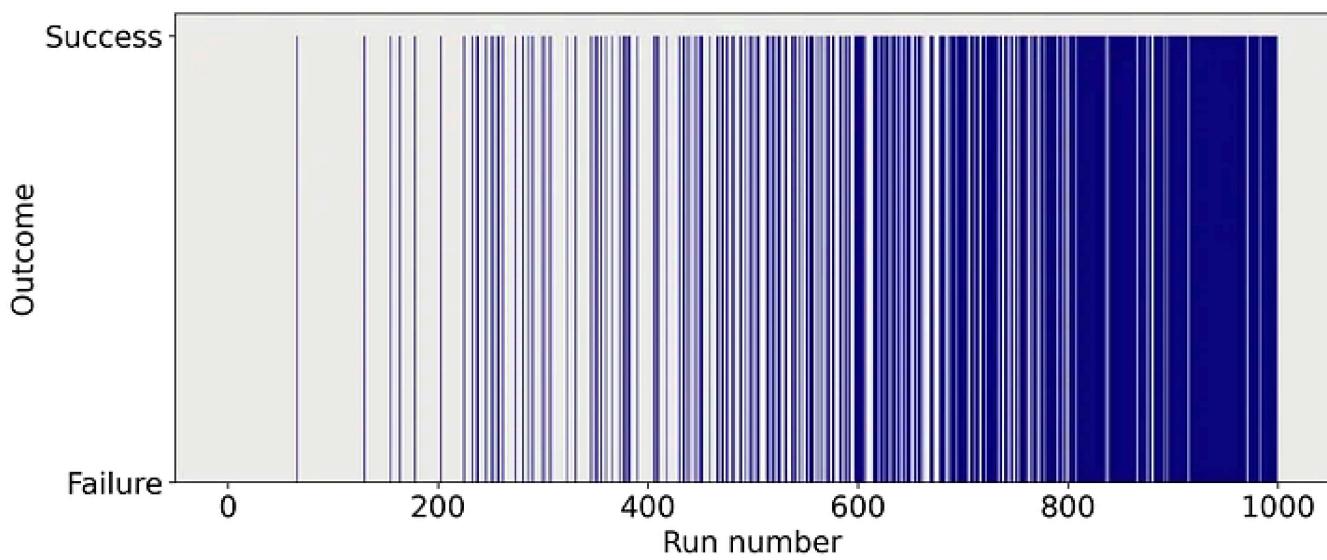


Image by author

Hey, the agent takes more time to consistently win the game now! And the Q-table has a lot more non-zero values than the previous one, which means the agent has learned several sequences of actions to reach the goal. It is understandable, since

this new agent is forced to explore state-action pairs instead of always exploiting ones with non-zero values.

Let's see if it's as successful as the previous one to win the game. In evaluation mode, we don't want exploration anymore because the agent is trained now.

```
episodes = 100
nb_success = 0

# Evaluation
for _ in range(100):
    state = environment.reset()
    done = False

    # Until the agent gets stuck or reaches the goal, keep training it
    while not done:
        # Choose the action with the highest value in the current state
        action = np.argmax(qtable[state])

        # Implement this action and move the agent in the desired direction
        new_state, reward, done, info = environment.step(action)

        # Update our current state
        state = new_state

        # When we get a reward, it means we solved the game
        nb_success += reward

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

Success rate = 100.0%

Phew, it's another **100% success rate!** We didn't degrade the model. 😊 The benefits of this approach might not be obvious in this example, but our model became **less static and more flexible**. It learned different paths (sequences of actions) from S to G instead of just one as in the previous approach. More exploration can degrade performance but it's necessary to train agents that can adapt to new environments.

IV. Challenge: slippery Frozen Lake

We didn't solve the entire Frozen Lake environment: we only trained an agent on the non-slippery version, using `is_slippery = False` during initialization. In the slippery variant, the action the agent takes only has 33% chance of succeeding. In case of failure, one of the three other actions is randomly taken instead. This feature adds a lot of randomness to the training, which makes things more difficult for our agent. Let's see how well our code is doing in this new environment...

```
environment = gym.make("FrozenLake-v1", is_slippery=True)
environment.reset()

# We re-initialize the Q-table
qtable = np.zeros((environment.observation_space.n, environment.action_space.n))

# Hyperparameters
episodes = 1000          # Total number of episodes
alpha = 0.5                # Learning rate
gamma = 0.9                # Discount factor
epsilon = 1.0              # Amount of randomness in the action selection
epsilon_decay = 0.001       # Fixed amount to decrease

# List of outcomes to plot
outcomes = []

print('Q-table before training:')
print(qtable)

# Training
for _ in range(episodes):
    state = environment.reset()
    done = False

    # By default, we consider our outcome to be a failure
    outcomes.append("Failure")

    # Until the agent gets stuck in a hole or reaches the goal, keep training i
    while not done:
        # Generate a random number between 0 and 1
        rnd = np.random.random()

        # If random number < epsilon, take a random action
        if rnd < epsilon:
            action = environment.action_space.sample()
        # Else, take the action with the highest value in the current state
        else:
            action = np.argmax(qtable[state])

        # Implement this action and move the agent in the desired direction
        new_state, reward, done, info = environment.step(action)
```

```

# Update Q(s,a)
qtable[state, action] = qtable[state, action] + \
                        alpha * (reward + gamma * np.max(qtable[new_st
                        

# Update our current state
state = new_state

# If we have a reward, it means that our outcome is a success
if reward:
    outcomes[-1] = "Success"

# Update epsilon
epsilon = max(epsilon - epsilon_decay, 0)

print()
print('=====')
print('Q-table after training:')
print(qtable)

# Plot outcomes
plt.figure(figsize=(12, 5))
plt.xlabel("Run number")
plt.ylabel("Outcome")
ax = plt.gca()
ax.set_facecolor('#efeeea')
plt.bar(range(len(outcomes)), outcomes, color="#0A047A", width=1.0)
plt.show()

episodes = 100
nb_success = 0

# Evaluation
for _ in range(episodes):
    state = environment.reset()
    done = False

    # Until the agent gets stuck or reaches the goal, keep training it
    while not done:
        # Choose the action with the highest value in the current state
        action = np.argmax(qtable[state])

        # Implement this action and move the agent in the desired direction
        new_state, reward, done, info = environment.step(action)

        # Update our current state
        state = new_state

        # When we get a reward, it means we solved the game
        nb_success += reward

    # Let's check our success rate!
    print(f"Success rate = {nb_success/episodes*100}%")

```

Q-table before training:

```
[[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. 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. 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. 0.]  
 [0. 0. 0. 0.]  
 [0. 0. 0. 0.]
```

=====

Q-table after training:

```
[[0.06208723 0.02559574 0.02022059 0.01985828]  
 [0.01397208 0.01425862 0.01305446 0.03333396]  
 [0.01318348 0.01294602 0.01356014 0.01461235]  
 [0.01117016 0.00752795 0.00870601 0.01278227]  
 [0.08696239 0.01894036 0.01542694 0.02307306]  
 [0. 0. 0. ]  
 [0.09027682 0.00490451 0.00793372 0.00448314]  
 [0. 0. 0. ]  
 [0.03488138 0.03987256 0.05172554 0.10780482]  
 [0.12444437 0.12321815 0.06462294 0.07084008]  
 [0.13216145 0.09460133 0.09949734 0.08022573]  
 [0. 0. 0. ]  
 [0. 0. 0. ]  
 [0.1606242 0.18174032 0.16636549 0.11444442]  
 [0.4216631 0.42345944 0.40825367 0.74082329]  
 [0. 0. 0. ]]
```

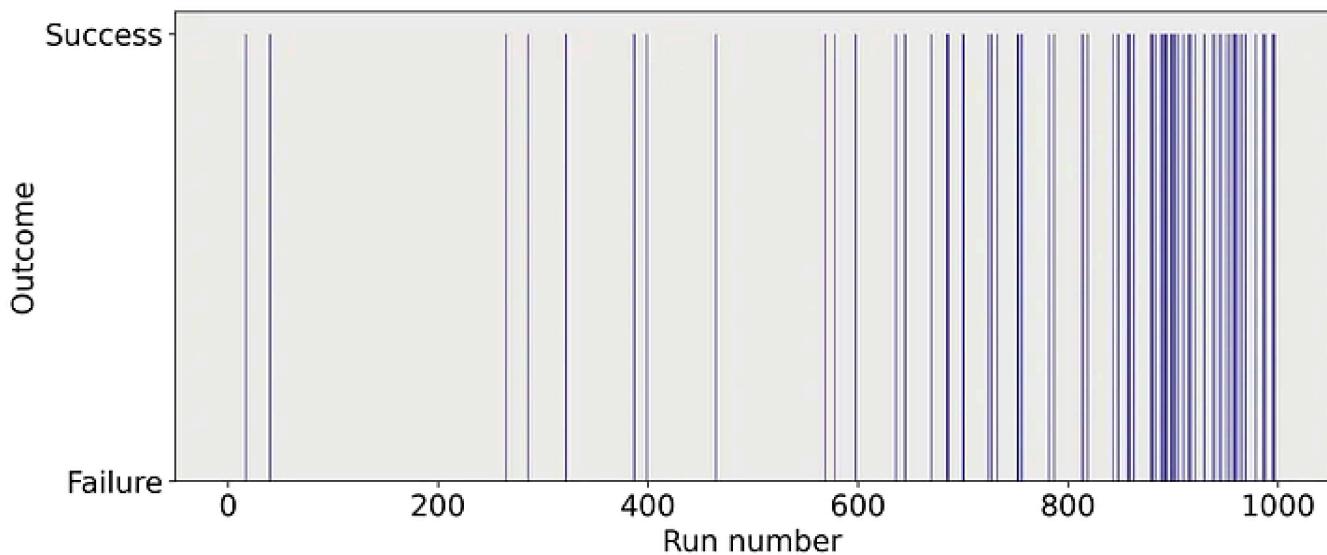


Image by author

Success rate = 17.0%

Of it's not so good. But can you improve the performance by tweaking the different parameters we talked about? I encourage you to take this little challenge and do it on your own to have fun with reinforcement learning and check if you understood everything we said about Q-learning. And why not implementing exponential decay for the epsilon-greedy algorithm too? During this quick exercise, you might realise that slightly modifying the hyperparameters can completely destroy the results. This is another quirk of reinforcement learning: hyperparameters are quite moody, and it is important to understand their meaning if you want to tweak them. It's always good to test and try new combinations to **build your intuition and become more efficient**. Good luck and have fun!

Conclusion

Q-learning is a simple yet powerful algorithm at the core of reinforcement learning. In this article,

- We learned to **interact with the `gym` environment** to choose actions and move our agent
- We introduced the idea of a **Q-table**, where rows are states, columns are actions, and cells are the value of an action in a given state
- We experimentally recreated the **Q-learning update formula** to tackle the sparse reward problem

- We implemented an entire training and evaluation process, that solved the Frozen Lake environment with 100% success rate
- We implemented the famous **epsilon-greedy algorithm** in order to create a tradeoff between the exploration of unknown state-action pairs and the exploitation of the most successful ones

The Frozen Lake is a very simple environment, but others can have **so many states and actions that it becomes impossible to store the Q-table in memory**. This is especially the case in environments where events are not discrete, but continuous (like Super Mario Bros. or Minecraft). When the problem arises, a popular technique consists of training a deep neural network to approximate the Q-table. This method adds several layers of complexity, since the neural networks are not very stable.

Until then, share this article if it helped you and [follow me on Twitter](#) and [Medium](#) for more practical content around machine learning and deep learning.

Machine Learning

Reinforcement Learning

Programming

Tutorial

Deep Dives

Some rights reserved 

tds

Follow

Published in Towards Data Science

768K Followers · Last published 4 hours ago

Your home for data science and AI. The world's leading publication for data science, data analytics, data engineering, machine learning, and artificial intelligence professionals.

[Follow](#)

Written by Maxime Labonne

8.2K Followers · 13 Following

Ph.D., Staff ML Scientist @ Liquid AI • Author of "Hands-On Graph Neural Networks" • x.com/maximelabonne

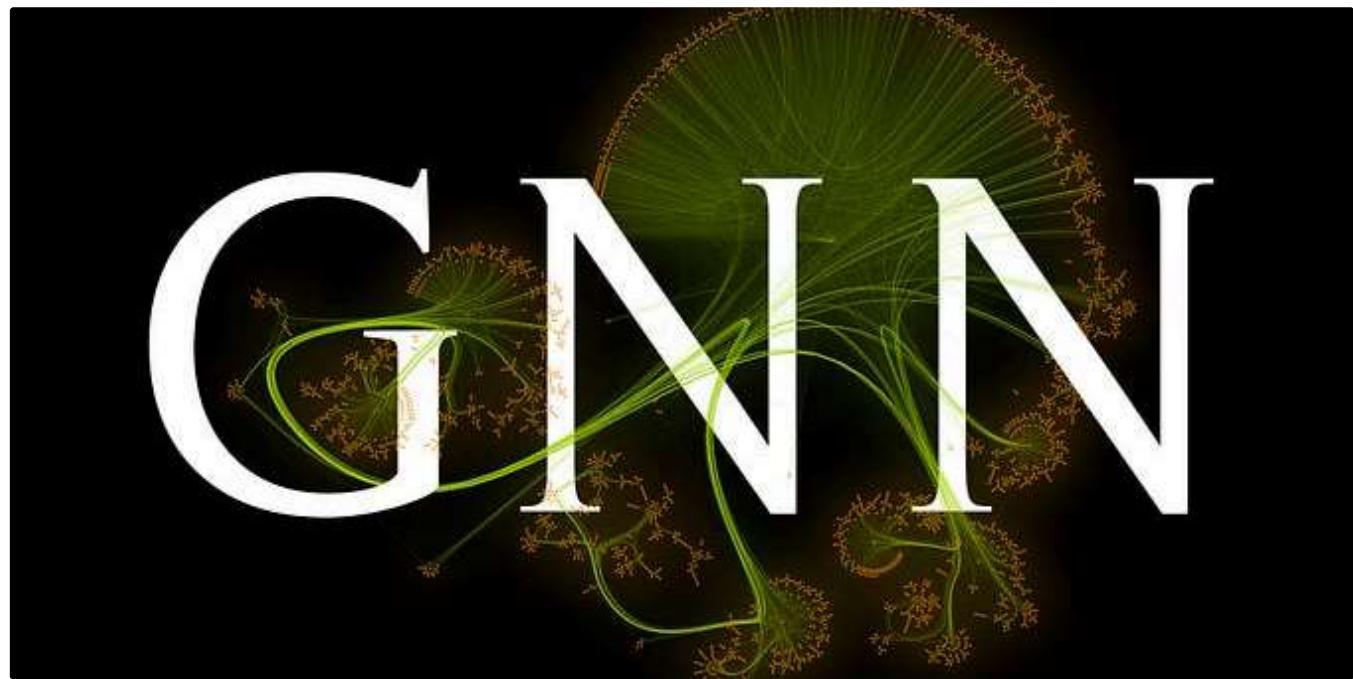
[Open in app](#) ↗

[Sign up](#)[Sign in](#)

Medium



Search



 In Towards Data Science by Maxime Labonne 

Graph Convolutional Networks: Introduction to GNNs

A step-by-step guide using PyTorch Geometric

Aug 14, 2023

1.4K

8



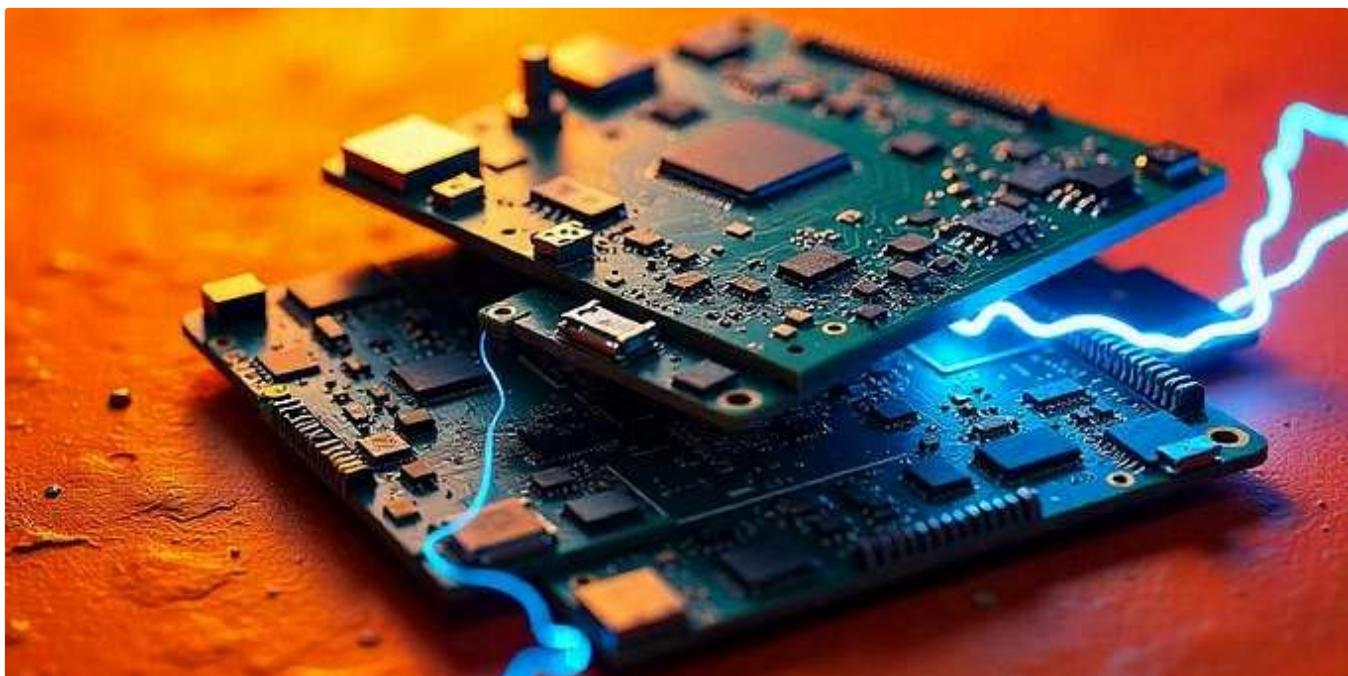


 In Towards Data Science by Jonathan R. Williford, PhD

Neuromorphic Computing—an Edgier, Greener AI

There are periodic proclamations of the coming neuromorphic computing revolution. While there remain substantial challenges in the field...

6d ago  37



 In Towards Data Science by Bradney Smith

Mistral 7B Explained: Towards More Efficient Language Models

RMS Norm, RoPE, GQA, SWA, KV Cache, and more!

1d ago

257

1

 In Towards Data Science by Maxime Labonne 

Efficiently iterating over rows in a Pandas DataFrame

Never use iterrows and itertuples again

Mar 21, 2022

1.3K

8

[See all from Maxime Labonne](#)[See all from Towards Data Science](#)

Recommended from Medium



 In Towards Data Science by Cristian Leo

Reinforcement Learning 101: Q-Learning

Decoding the Math behind Q-Learning, Action-Value Functions, Bellman Equations, and building them from scratch in Python.

★ Feb 28 ⌘ 478 🗣 6



 In Lyft Engineering by Jonas Timmermann

Lyft's Reinforcement Learning Platform

Tackling decision making problems with a platform for developing & serving Reinforcement Learning models with a focus on Contextual Bandits

Mar 12 152

Lists



Predictive Modeling w/ Python

20 stories · 1688 saves



General Coding Knowledge

20 stories · 1776 saves



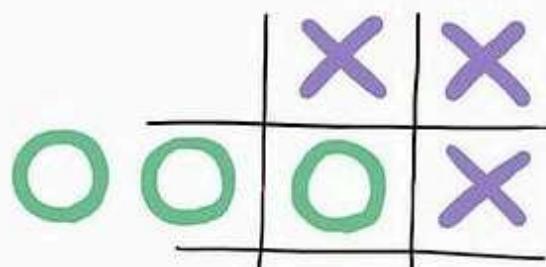
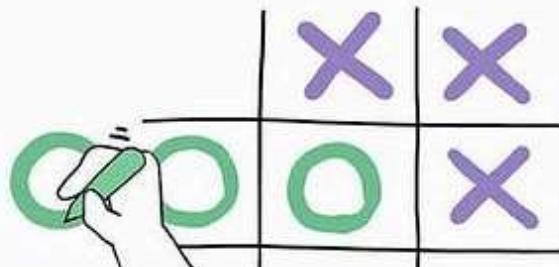
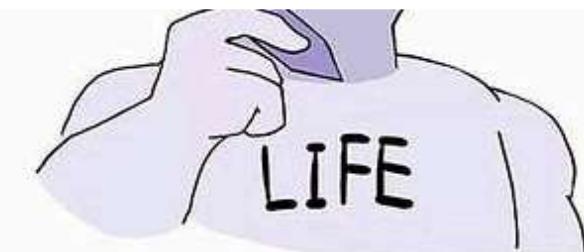
Practical Guides to Machine Learning

10 stories · 2055 saves



Natural Language Processing

1832 stories · 1459 saves



Amit Yadav

Reinforcement Learning in Games: A Complete Guide

Not a Medium member? Read the full story by [clicking here](#).

Jun 26 21



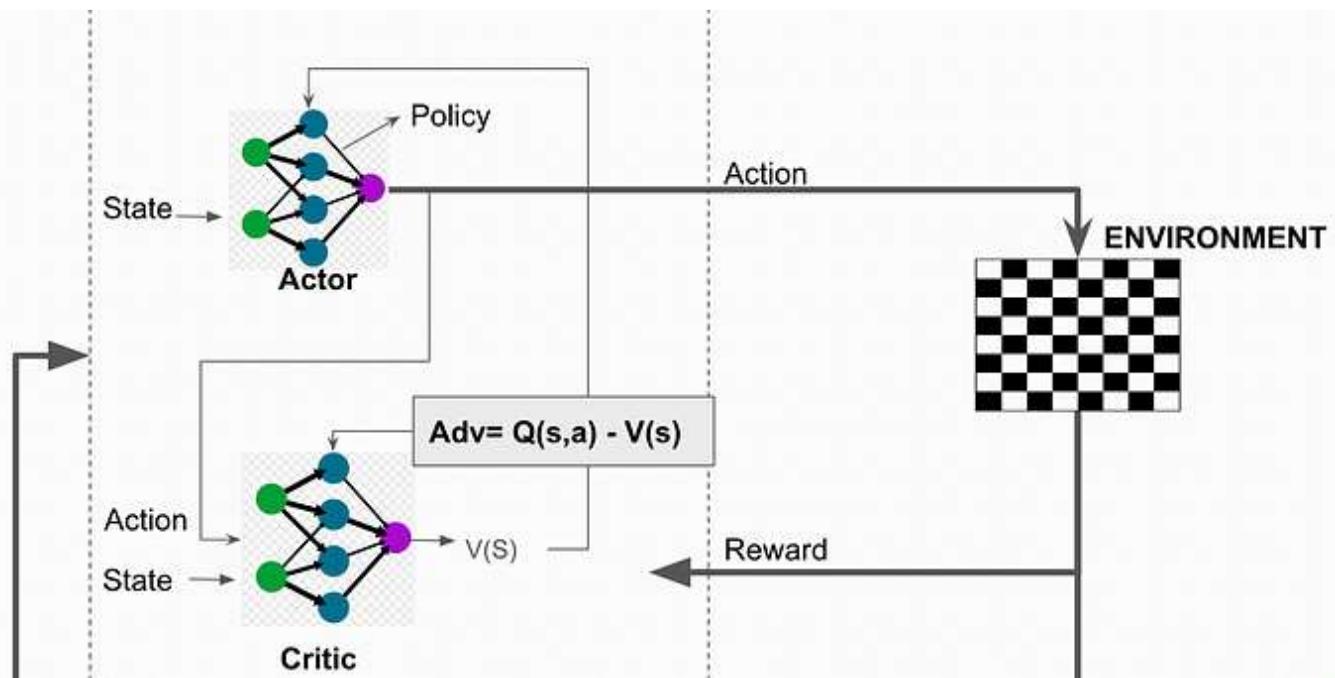


 Haitham Bou Ammar

From MCTS to Alpha-Zero with PyTorch—Part I (Building a Tic-Tac-Toe'r)

AlphaZero is a deep reinforcement learning algorithm developed by DeepMind that has achieved superhuman performance in games like Chess...

Sep 9  1

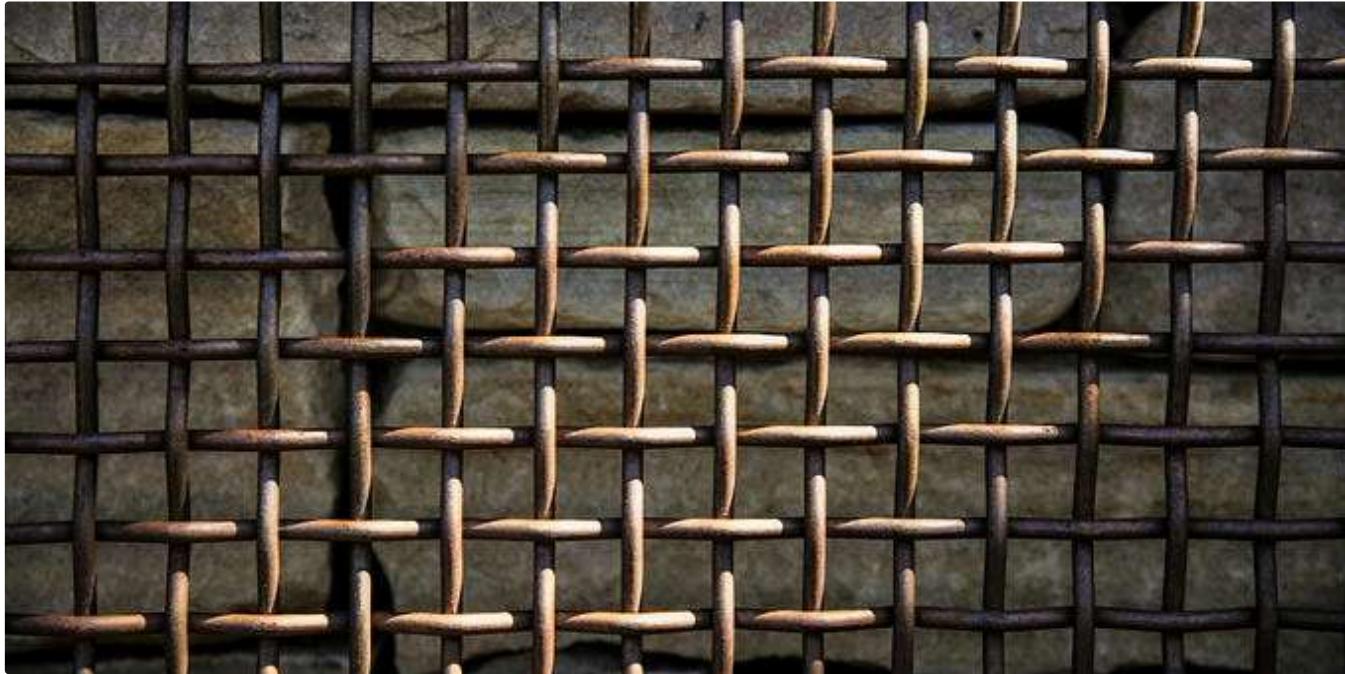


 Renu Khandelwal

Unlocking the Secrets of Actor-Critic Reinforcement Learning: A Beginner's Guide

Understanding Actor-Critic Mechanisms, Different Flavors of Actor-Critic Algorithms, and a Simple Implementation in PyTorch

Feb 21, 2023 188 3



Vitality Learning

Solving the Gridworld Problem Using Reinforcement Learning in Python

Reinforcement Learning (RL) is an exciting and powerful paradigm that allows agents to learn optimal behaviors through trial and error. In...

Oct 14 1



See more recommendations