What is Q-Learning?

Q-learning is a model-free, value-based, off-policy algorithm that will find the best series of actions based on the agent's current state. The “Q” stands for quality. Quality represents how valuable the action is in maximizing future rewards.

The **model-based** algorithms use transition and reward functions to estimate the optimal policy and create the model. In contrast, **model-free** algorithms learn the consequences of their actions through the experience without transition and reward function.

The **value-based** method trains the value function to learn which state is more valuable and take action. On the other hand, **policy-based** methods train the policy directly to learn which action to take in a given state.

In the **off-policy**, the algorithm evaluates and updates a policy that differs from the policy used to take an action. Conversely, the **on-policy** algorithm evaluates and improves the same policy used to take an action.

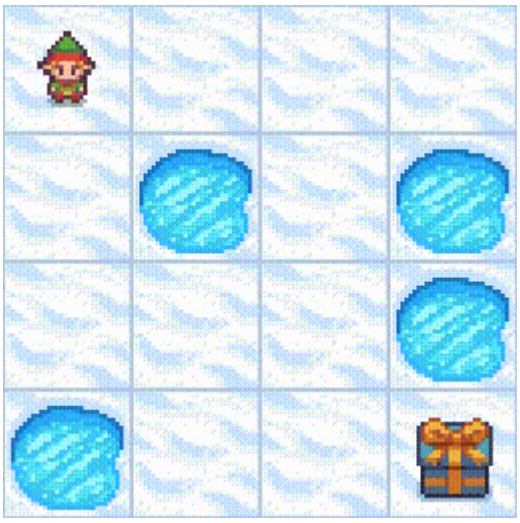
Key Terminologies in Q-learning

Before we jump into how Q-learning works, we need to learn a few useful terminologies to understand Q-learning's fundamentals.

- **States(s)**: the current position of the agent in the environment.
- **Action(a)**: a step taken by the agent in a particular state.
- **Rewards**: for every action, the agent receives a reward and penalty.
- **Episodes**: the end of the stage, where agents can’t take new action. It happens when the agent has achieved the goal or failed.
- **$Q(S_{t+1}, a)$**: expected optimal Q-value of doing the action in a particular state.
- **$Q(S_t, A_t)$**: it is the current estimation of $Q(S_{t+1}, a)$.
- **Q-Table**: the agent maintains the Q-table of sets of states and actions.
- **Temporal Differences(TD)**: used to estimate the expected value of $Q(S_{t+1}, a)$ by using the current state and action and previous state and action.

How Does Q-Learning Work?

We will learn in detail how Q-learning works by using the example of a frozen lake. In this environment, the agent must cross the frozen lake from the start to the goal, without falling into the holes. The best strategy is to reach goals by taking the shortest path.



Q-Table

The agent will use a Q-table to take the best possible action based on the expected reward for each state in the environment. In simple words, a Q-table is a data structure of sets of actions and states, and we use the Q-learning algorithm to update the values in the table.

Q-Function

The Q-function uses the Bellman equation and takes state(s) and action(a) as input. The equation simplifies the state values and state-action value calculation.

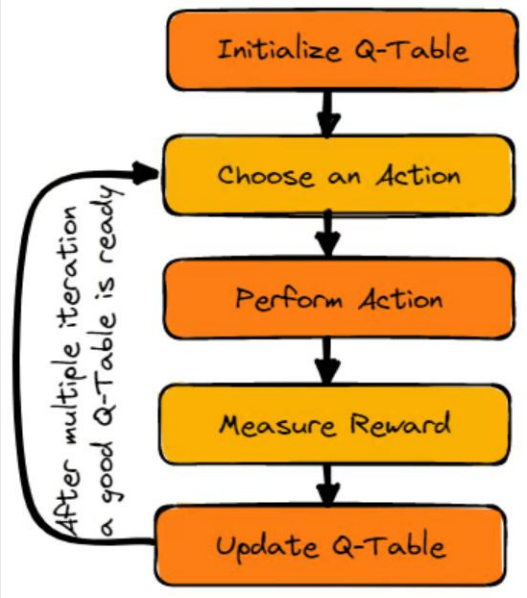
Q-learning algorithm

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

Q-Values for the state given a particular state

Expected discounted cumulative reward

Given the state and action



Initialize Q-Table

We will first initialize the Q-table. We will build the table with columns based on the number of actions and rows based on the number of states.

In our example, the character can move up, down, left, and right. We have four possible actions and four states(start, Idle, wrong path, and end). You can also consider the wrong path for falling into the hole. We will initialize the Q-Table with values at 0.

	→	←	↑	↓
Start	0	0	0	0
Idle	0	0	0	0
Hole	0	0	0	0
End	0	0	0	0

Choose an Action

The second step is quite simple. At the start, the agent will choose to take the random action(down or right), and on the second run, it will use an updated Q-Table to select the action.

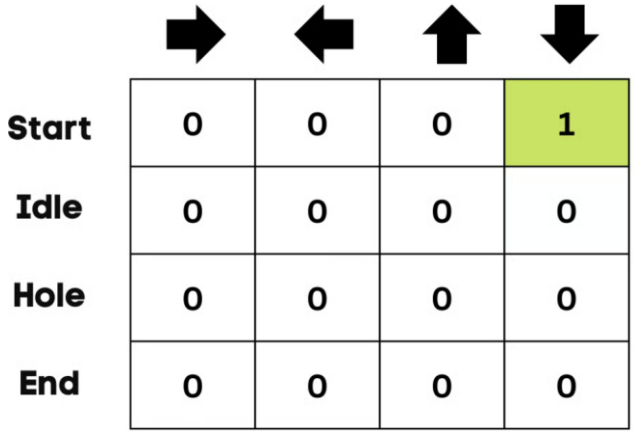
Perform an Action

Choosing an action and performing the action will repeat multiple times until the training loop stops. The first action and state are selected using the Q-Table. In our case, all values of the Q-Table are zero.

Then, the agent will move down and update the Q-Table using the Bellman equation. With every move, we will be updating values in the Q-Table and also using it for determining the best course of action.

Initially, the agent is in exploration mode and chooses a random action to explore the environment. The Epsilon Greedy Strategy is a simple method to balance exploration and exploitation. The epsilon stands for the probability of choosing to explore and exploits when there are smaller chances of exploring.

At the start, the epsilon rate is higher, meaning the agent is in exploration mode. While exploring the environment, the epsilon decreases, and agents start to exploit the environment. During exploration, with every iteration, the agent becomes more confident in estimating Q-values



	→	←	↑	↓
Start	0	0	0	1
Idle	0	0	0	0
Hole	0	0	0	0
End	0	0	0	0

In the frozen lake example, the agent is unaware of the environment, so it takes random action (move down) to start. As we can see in the above image, the Q-Table is updated using the Bellman equation.

Measuring the Rewards

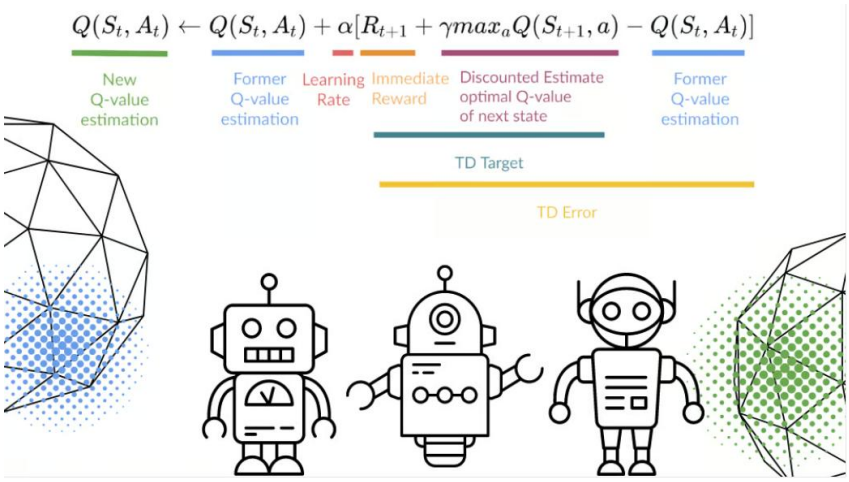
After taking the action, we will measure the outcome and the reward.

- The reward for reaching the goal is +1
- The reward for taking the wrong path (falling into the hole) is 0
- The reward for Idle or moving on the frozen lake is also 0.

Update Q-Table

We will update the function $Q(S_t, A_t)$ using the equation. It uses the previous episode's estimated Q-values, learning rate, and Temporal Differences error. Temporal Differences error is calculated using Immediate reward, the discounted maximum expected future reward, and the former estimation Q-value.

The process is repeated multiple times until the Q-Table is updated and the Q-value function is maximized.



At the start, the agent is exploring the environment to update the Q-table. And when the Q-Table is ready, the agent will start exploiting and start taking better decisions.

	➡	⬅	⬆	⬇
Start	0	1	0	0
Idle	2	0	0	3
Hole	0	2	0	0
End	1	0	0	0

n the case of a frozen lake, the agent will learn to take the shortest path to reach the goal and avoid jumping into the holes.

Q-Learning Python Tutorial

In this section, we will build our Q-learning model from scratch using the Gym environment, Pygame, and Numpy. It includes initializing the environment and Q-Table, defining greedy policy, setting hyperparameters, creating and running the training loop and evaluation, and visualizing the results.

Setup a Virtual Display
To generate a replay video of agent and environment.
"""

Commented out IPython magic to ensure Python compatibility.
%%capture
!pip install pyglet==1.5.1
!apt install python-opengl
!apt install ffmpeg
!apt install xvfb
!pip3 install pyvirtualdisplay
#
Virtual display
from pyvirtualdisplay import Display
#
virtual_display = Display(visible=0, size=(1400, 900))
virtual_display.start()

"""## Install dependencies"""

Commented out IPython magic to ensure Python compatibility.
%%capture
!pip install gym==0.24
!pip install pygame
!pip install numpy
#
!pip install imageio imageio_ffmpeg

"""## Import the packages"""

import numpy as np
import gym
import random
import imageio
from tqdm.notebook import trange

"""## Frozen Lake"""

Create the FrozenLake-v1 environment using 4x4 map and non-slippery version

```
env = gym.make("FrozenLake-v1",map_name="4x4",is_slippery=False)

"""### Understanding the FrozenLake environment"""

print("_____OBSERVATION SPACE_____ \n")
print("Observation Space", env.observation_space)
print("Sample observation", env.observation_space.sample()) # Get a random observation

print("\n _____ACTION SPACE_____ \n")
print("Action Space Shape", env.action_space.n)
print("Action Space Sample", env.action_space.sample()) # Take a random action

"""## Create and Initialize the Q-table"""

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

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

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

Qtable_frozenlake = initialize_q_table(state_space, action_space)

"""## Define the epsilon-greedy policy"""

def epsilon_greedy_policy(Qtable, state, epsilon):
  # Randomly generate a number between 0 and 1
  random_int = random.uniform(0,1)
  # if random_int > greater than epsilon --> exploitation
  if random_int > epsilon:
    # Take the action with the highest value given a state
    # np.argmax can be useful here
    action = np.argmax(Qtable[state])
  # else --> exploration
  else:
    action = env.action_space.sample()

  return action

"""## Define the greedy policy"""
```

```
def greedy_policy(Qtable, state):
  # Exploitation: take the action with the highest state, action value
  action = np.argmax(Qtable[state])

  return action

"""## Define the hyperparameters"""

# Training parameters
n_training_episodes = 10000  # Total training episodes
learning_rate = 0.7          # Learning rate

# Evaluation parameters
n_eval_episodes = 100        # Total number of test episodes

# Environment parameters
env_id = "FrozenLake-v1"     # Name of the environment
max_steps = 99               # Max steps per episode
gamma = 0.95                 # Discounting rate
eval_seed = []               # The evaluation seed of the environment

# Exploration parameters
max_epsilon = 1.0            # Exploration probability at start
min_epsilon = 0.05           # Minimum exploration probability
decay_rate = 0.0005          # Exponential decay rate for exploration prob

"""## Training the model"""

def train(n_training_episodes, min_epsilon, max_epsilon, decay_rate, env, max_steps, Qtable):
  for episode in trange(n_training_episodes):
    # Reduce epsilon (because we need less and less exploration)
    epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.exp(-decay_rate*episode)
    # Reset the environment
    state = env.reset()
    step = 0
    done = False

    # repeat
    for step in range(max_steps):
      # Choose the action At using epsilon greedy policy
      action = epsilon_greedy_policy(Qtable, state, epsilon)

      # Take action At and observe Rt+1 and St+1
      # Take the action (a) and observe the outcome state(s') and reward (r)
      new_state, reward, done, info = env.step(action)
```

```
      # Update Q(s,a):= Q(s,a) + lr [R(s,a) + gamma * max Q(s',a') - Q(s,a)]
      Qtable[state][action] = Qtable[state][action] + learning_rate * (reward + gamma *
np.max(Qtable[new_state]) - Qtable[state][action])

      # If done, finish the episode
      if done:
        break

      # Our state is the new state
      state = new_state
  return Qtable

Qtable_frozenlake = train(n_training_episodes, min_epsilon, max_epsilon, decay_rate, env,
max_steps, Qtable_frozenlake)

"""## Tranined Q-Learning table"""

Qtable_frozenlake

"""## Model evaluation"""

def evaluate_agent(env, max_steps, n_eval_episodes, Q, seed):
  """
  Evaluate the agent for ``n_eval_episodes`` episodes and returns average reward and std of
reward.
  :param env: The evaluation environment
  :param n_eval_episodes: Number of episode to evaluate the agent
  :param Q: The Q-table
  :param seed: The evaluation seed array (for taxi-v3)
  """
  episode_rewards = []
  for episode in range(n_eval_episodes):
    if seed:
      state = env.reset(seed=seed[episode])
    else:
      state = env.reset()
    step = 0
    done = False
    total_rewards_ep = 0

    for step in range(max_steps):
      # Take the action (index) that have the maximum expected future reward given that state
      action = np.argmax(Q[state][:])
      new_state, reward, done, info = env.step(action)
      total_rewards_ep += reward
```

```
      if done:
        break
      state = new_state
    episode_rewards.append(total_rewards_ep)
  mean_reward = np.mean(episode_rewards)
  std_reward = np.std(episode_rewards)

  return mean_reward, std_reward

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

"""## Visualizing the results"""

def record_video(env, Qtable, out_directory, fps=1):
  images = []
  done = False
  state = env.reset(seed=random.randint(0,500))
  img = env.render(mode='rgb_array')
  images.append(img)
  while not done:
    # Take the action (index) that have the maximum expected future reward given that state
    action = np.argmax(Qtable[state][:])
    state, reward, done, info = env.step(action) # We directly put next_state = state for recording
logic
    img = env.render(mode='rgb_array')
    images.append(img)
  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"""
video_path="/content/replay.gif"
video_fps=1
record_video(env, Qtable_frozenlake, video_path, video_fps)
from IPython.display import Image
Image('./replay.gif')

**********************************************************************************
```