Frozen Lake with Q-Learning

Understanding the state and action space

- Link to study the environment: https://gymnasium.farama.org/environments/toy_text/frozen_lake/

```
import gymnasium as gym

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

print(f"The possible state space: {env.observation_space.n}")
print(f"The possible action space: {env.action_space.n}")
```

```
The possible state space: 16
The possible action space: 4
```

Create a Q-Learning Table

```
import numpy as np

# A function that creates Q Table based on the size of state space and action space size.
def initialize_q_table(state_space, action_space):
  """
  A function that creates Q Table based on the size of state space and action space size.

  Args:
    state_space (int): Number of available unique state in the state space.
    action_space (int): Number of available actions in the state space.

  Returns:
    q_table (float array): A Q Table with the shape of (state_space, action_space).

  """
  q_table = np.zeros((state_space, action_space))
  return q_table

```

Create and test the Q-Learning Table

```
state_space = env.observation_space.n
action_space = env.action_space.n

QTable = initialize_q_table(state_space, action_space)
print(f"The shape of the Q Table is: {QTable.shape}")
```

```
The shape of the Q Table is: (16, 4)
```

Define Greedy Policy

- Always take the action that has the highest value.

```
def greedy_policy(q_table, state):
  """
  Greedy Policy - always take the action that has the highest q value within a state.

  Args:
    q_table (float array): A Q Table that has the size of (state_space, action_space).
    state (int): The current state that the agent is in.

  Returns:
    action (int): The action to be taken by the agent under the greed_policy.
  """
  action = np.argmax(q_table[state])
  return action
```

Define Epsilon-Greedy Policy

- The agent has a probability of ε in taking a random action (Exploration) and a probability of 1-ε in following Greedy Policy (Exploitation).

```
def epsilon_greedy_policy(q_table, state, epsilon, env):

  """
  Epsilon Greedy Policy - The agent has a probability of ε in taking a random action (Exploration) and a probability of 1-ε in follo

  Args:
    q_table (float array): A Q Table that has the size of (state_space, action_space).
    state (int): The current state that the agent is in.
    epsilon (float): A number that decides if exploration or exploitation.
    env (gymnaisum.env): The environment that the the agent is in.

  Returns:
    action (int): The action to be taken by the agent under the greed_policy.
  """

  # Sample a random number
  probability = np.random.uniform(0, 1)

  # Exploitation - Follow Greedy Policy
  if probability > epsilon:
    action = greedy_policy(q_table, state)
  # Exploration - Take random action
  else:
    action = env.action_space.sample()

  return action

```

Defining 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
```

Create the training loop


```
from tqdm.auto import tqdm

def train(n_training_episodes, max_steps, env, q_table, epsilon_min, epsilon_max, decay_rate, lr, gamma):

  """
  Training Loop for a Q-Learning Agent.

  Args:
    n_training_episodes (int): Number of episodes to be trained.
    max_steps (int): Maximum number of steps per episodes.
    env (gymnaisum.env): The environment that the the agent is in.
    q_table (float array): A Q Table that has the size of (state_space, action_space).
    epsilon_min (float): Lower bound for the epsilon value, expected between range of 0 to 1.
    epsilon_max (float): Upper bound for the epsilon value, expected between range of 0 to 1.
    decay_rate (float): Decay rate for the epsilon value.
    lr (float): Learning rate for the agent.
    gamma (float): Discount factor for the reward.

  Returns:
    q_table (float array): A trained Q Table that has the size of (state_space, action_space).
  """


  for episode in tqdm(range(n_training_episodes)):
    # Reset status of termination or truncation
    terminated = False
    truncated = False

    # Reset state
    state, info = env.reset()

    # Adjust epsilon
    epsilon = epsilon_min + (epsilon_max - epsilon_min) * np.exp(-decay_rate * episode)

    # Begin a new episode step by step
    for step in range(max_steps):

      # Sample an action based on epsilon_greedy_policy
      action = epsilon_greedy_policy(q_table, state, epsilon, env)

      # Perform the sampled action and observe the new state and received reward
      new_state, reward, terminated, truncated, info = env.step(action)

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

      # Check if reached end of episodes
      if terminated or truncated:
        break

      # Update the state
      state = new_state

  return q_table
```

Perform training

```
QTable_frozen_lake = train(n_training_episodes = n_training_episodes,
                           max_steps = max_steps,
                           env = env,
                           q_table = QTable,
                           epsilon_min = min_epsilon,
                           epsilon_max = max_epsilon,
                           decay_rate = decay_rate,
                           lr = learning_rate,
                           gamma = gamma)
```

```
100%    10000/10000 [00:02<00:00, 5055.18it/s]
```

Create evaluation code

```
def evaluate_agent(env, max_steps, n_eval_episodes, q_table, seed):

  """
  Evaluation for a Q-Learning Agent.

  Args:
    n_eval_episodes (int): Number of episodes to be evaluated.
    max_steps (int): Maximum number of steps per episodes.
    env (gymnaisum.env): The environment that the the agent is in.
    q_table (float array): A Q Table that has the size of (state_space, action_space).
    seed (int list): A list that consists of the initial state of the environment.

  Returns:
    mean_reward (float): Mean reward received over the evaluated episodes.
    std_reward (float): Standard deviation reward receiveed over the evaluated episodes.
  """

  # Initialize a list to store reward from each episode
  episode_rewards = []

  # Begin each episode
  for episode in tqdm(range(n_eval_episodes)):

    # Reset the environment parameters
    if seed:
      state, info = env.reset(seed = seed[episode])
    else:
      state, info = env.reset()
    terminated = False
    truncated = False

    # Reset the reward per episode
    reward_per_ep = 0

    # Repeat steps
    for step in range(max_steps):

      # Sample an action using greedy_policy and step with that
      action = greedy_policy(q_table, state)
      state, reward, terminated, truncated, info = env.step(action)

      # Add the reward
      reward_per_ep += reward

      # Check if terminated or truncated
      if terminated or truncated:
        break

    # Append the episode for this episode to the list
    episode_rewards.append(reward_per_ep)

  # End of all episodes - Calculate mean and standard deviation of reward
  mean_reward = np.mean(episode_rewards).item()
  std_reward = np.std(episode_rewards).item()

  return mean_reward, std_reward

```

Perform evaluation

```
mean_reward, std_reward = evaluate_agent(n_eval_episodes = n_eval_episodes,
                                         max_steps = max_steps,
                                         env = env,
                                         q_table = QTable_frozen_lake,
                                         seed = None)

print(f"The trained agent's performance: {mean_reward:.2f} +/- {std_reward:.2f}")
```

100% 100/100 [00:00<00:00, 4137.25it/s]

```
The trained agent's performance: 1.00 +/- 0.00
```

Functions made to push to Hugging Face Hub

- This code is provided by the tutorial: https://colab.research.google.com/github/huggingface/deep-rl-class/blob/master/notebooks/unit2/unit2.ipynb#scrollTo=paOynXy3aoJW
- **package_to_hub** cannot be used to push models to hub because the archictecture used here is custom made and not from stable-baselines3 as in Unit 1.

```
# Import Libary

from huggingface_hub import HfApi, snapshot_download
from huggingface_hub.repocard import metadata_eval_result, metadata_save

from pathlib import Path
import datetime
import json

import imageio
import random
import pickle
```

```
# A function made to record an episode to sbe showcased on the model card.

def record_video(env, Qtable, out_directory, fps=1):
  """
  Generate a replay video of the agent
  :param env
  :param Qtable: Qtable of our agent
  :param out_directory
  :param fps: how many frame per seconds (with taxi-v3 and frozenlake-v1 we use 1)
  """
  images = []
  terminated = False
  truncated = False
  state, info = env.reset(seed=random.randint(0,500))
  img = env.render()
  images.append(img)
  while not terminated or truncated:
    # Take the action (index) that have the maximum expected future reward given that state
    action = np.argmax(Qtable[state][:])
    state, reward, terminated, truncated, info = env.step(action) # We directly put next_state = state for recording logic
    img = env.render()
    images.append(img)
  imageio.mimsave(out_directory, [np.array(img) for i, img in enumerate(images)], fps=fps)
```

```
# Push to Hub

def push_to_hub(
    repo_id, model, env, video_fps=1, local_repo_path="hub"
):
    """
    Evaluate, Generate a video and Upload a model to Hugging Face Hub.
    This method does the complete pipeline:
    - It evaluates the model
    - It generates the model card
    - It generates a replay video of the agent
    - It pushes everything to the Hub

    :param repo_id: repo_id: id of the model repository from the Hugging Face Hub
    :param env
    :param video_fps: how many frame per seconds to record our video replay
    (with taxi-v3 and frozenlake-v1 we use 1)
    :param local_repo_path: where the local repository is
    """
    _, repo_name = repo_id.split("/")

    eval_env = env
    api = HfApi()

    # Step 1: Create the repo
    repo_url = api.create_repo(
        repo_id=repo_id,
        exist_ok=True,
    )

    # Step 2: Download files
    repo_local_path = Path(snapshot_download(repo_id=repo_id))

    # Step 3: Save the model
    if env.spec.kwargs.get("map_name"):
        model["map_name"] = env.spec.kwargs.get("map_name")
        if env.spec.kwargs.get("is_slippery", "") == False:
```

```python
            model["slippery"] = False

    # Pickle the model
    with open((repo_local_path) / "q-learning.pkl", "wb") as f:
        pickle.dump(model, f)

    # Step 4: Evaluate the model and build JSON with evaluation metrics
    mean_reward, std_reward = evaluate_agent(
        eval_env, model["max_steps"], model["n_eval_episodes"], model["qtable"], model["eval_seed"]
    )

    evaluate_data = {
        "env_id": model["env_id"],
        "mean_reward": mean_reward,
        "n_eval_episodes": model["n_eval_episodes"],
        "eval_datetime": datetime.datetime.now().isoformat()
    }

    # Write a JSON file called "results.json" that will contain the
    # evaluation results
    with open(repo_local_path / "results.json", "w") as outfile:
        json.dump(evaluate_data, outfile)

    # Step 5: Create the model card
    env_name = model["env_id"]
    if env.spec.kwargs.get("map_name"):
        env_name += "-" + env.spec.kwargs.get("map_name")

    if env.spec.kwargs.get("is_slippery", "") == False:
        env_name += "-" + "no_slippery"

    metadata = {}
    metadata["tags"] = [env_name, "q-learning", "reinforcement-learning", "custom-implementation"]

    # Add metrics
    eval = metadata_eval_result(
        model_pretty_name=repo_name,
        task_pretty_name="reinforcement-learning",
        task_id="reinforcement-learning",
        metrics_pretty_name="mean_reward",
        metrics_id="mean_reward",
        metrics_value=f"{mean_reward:.2f} +/- {std_reward:.2f}",
        dataset_pretty_name=env_name,
        dataset_id=env_name,
    )

    # Merges both dictionaries
    metadata = {**metadata, **eval}

    model_card = f"""
  # **Q-Learning** Agent playing **{env_id}**
  This is a trained model of a **Q-Learning** agent playing **{env_id}** .

  ## Usage

  ```python

  model = load_from_hub(repo_id="{repo_id}", filename="q-learning.pkl")

  env = gym.make(model["env_id"])
  ```
  """

    evaluate_agent(env, model["max_steps"], model["n_eval_episodes"], model["qtable"], model["eval_seed"])

    readme_path = repo_local_path / "README.md"
    readme = ""
    print(readme_path.exists())
    if readme_path.exists():
        with readme_path.open("r", encoding="utf8") as f:
            readme = f.read()
    else:
        readme = model_card

    with readme_path.open("w", encoding="utf-8") as f:
        f.write(readme)

    # Save our metrics to Readme metadata
    metadata_save(readme_path, metadata)

    # Step 6: Record a video
    video_path = repo_local_path / "replay.mp4"
```

```
    record_video(env, model["qtable"], video_path, video_fps)

    # Step 7. Push everything to the Hub
    api.upload_folder(
        repo_id=repo_id,
        folder_path=repo_local_path,
        path_in_repo=".",
    )

    print("Your model is pushed to the Hub. You can view your model here: ", repo_url)
```

```
# Create the dictionary that describe the model.

model = {
    "env_id": env_id,
    "max_steps": max_steps,
    "n_training_episodes": n_training_episodes,
    "n_eval_episodes": n_eval_episodes,
    "eval_seed": eval_seed,

    "learning_rate": learning_rate,
    "gamma": gamma,

    "max_epsilon": max_epsilon,
    "min_epsilon": min_epsilon,
    "decay_rate": decay_rate,

    "qtable": QTable_frozen_lake
}
```

Push to Hugging Face Hub

- Create a new token with write role here: https://huggingface.co/settings/tokens

```
from huggingface_hub import notebook_login

notebook_login()
```

```
push_to_hub(repo_id = "wengti0608/q-FrozenLake-v1-4x4-noSlippery",
            model = model,
            env = env)
```

```
Fetching 5 files: 100%      5/5 [00:00<00:00,  3.37it/s]
q-learning.pkl: 100%      915/915 [00:00<00:00, 1.37kB/s]
results.json: 100%      118/118 [00:00<00:00, 6.93kB/s]
100%      100/100 [00:00<00:00, 3321.25it/s]
100%      100/100 [00:00<00:00, 2850.90it/s]
True
Uploading...: 100%      915/915 [00:00<00:00, 2.54kB/s]
Your model is pushed to the Hub. You can view your model here:  https://huggingface.co/wengti0608/q-FrozenLake-v1-4x4-noSlippery
```

Taxi with Q-Learning

- Link to study the environment: https://gymnasium.farama.org/environments/toy_text/taxi/

Understanding the state and action space

```
import gymnasium as gym

env = gym.make("Taxi-v3",
               render_mode = 'rgb_array')

print(f"The possible state space: {env.observation_space.n}")
print(f"The possible action space: {env.action_space.n}")
```

```
The possible state space: 500
The possible action space: 6
```

Create a Q-Learning Table

```
state_space = env.observation_space.n
action_space = env.action_space.n

q_table = initialize_q_table(state_space, action_space)

print(f"The shape of the q_table: {q_table.shape}")
```

```
The shape of the q_table: (500, 6)
```

Define the hyperparameter

```
# Training parameters
n_training_episodes = 25000  # Total training episodes
learning_rate = 0.7           # Learning rate

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

# DO NOT MODIFY EVAL_SEED
eval_seed = [16,54,165,177,191,191,120,80,149,178,48,38,6,125,174,73,50,172,100,148,146,6,25,40,68,148,49,167,9,97,164,176,61,7,54,
 161,131,184,51,170,12,120,113,95,126,51,98,36,135,54,82,45,95,89,59,95,124,9,113,58,85,51,134,121,169,105,21,30,11,50,65,12,43,82,
 112,102,168,123,97,21,83,158,26,80,63,5,81,32,11,28,148] # Evaluation seed, this ensures that all classmates agents are trained on
                                                          # Each seed has a specific starting state

# Environment parameters
env_id = "Taxi-v3"           # Name of the environment
max_steps = 99 # 99               # Max steps per episode
gamma = 0.95                 # Discounting rate

# Exploration parameters
max_epsilon = 1.0             # Exploration probability at start
min_epsilon = 0.05           # Minimum exploration probability
decay_rate = 0.005            # Exponential decay rate for exploration prob
```

Perform Training

```
QTable_taxi = train(n_training_episodes = n_training_episodes,
                    max_steps = max_steps,
                    env = env,
                    q_table = q_table,
                    epsilon_min = min_epsilon,
                    epsilon_max = max_epsilon,
                    decay_rate = decay_rate,
                    lr = learning_rate,
                    gamma = gamma)

```

100% 25000/25000 [01:41<00:00, 243.04it/s]

Perform evaluation

```
mean_reward, std_reward = evaluate_agent(env = env,
                                         max_steps = max_steps,
                                         n_eval_episodes = n_eval_episodes,
                                         q_table = QTable_taxi,
                                         seed = eval_seed)

print(f"The agent has a performance of: {mean_reward:.2f} +/- {std_reward:.2f}")
```

100% 100/100 [00:00<00:00, 1637.52it/s]

```
The agent has a performance of: 7.56 +/- 2.71
```

Push to Hugging Face Hub

- Create a new token with write role here: https://huggingface.co/settings/tokens

```
from huggingface_hub import notebook_login
notebook_login()
```

Invalid user token.

```
# Create the dictionary that describe the model.

model = {
    "env_id": env_id,
    "max_steps": max_steps,
    "n_training_episodes": n_training_episodes,
    "n_eval_episodes": n_eval_episodes,
    "eval_seed": eval_seed,

    "learning_rate": learning_rate,
    "gamma": gamma,

    "max_epsilon": max_epsilon,
    "min_epsilon": min_epsilon,
    "decay_rate": decay_rate,

    "qtable": QTable_taxi
}
```

```
push_to_hub(repo_id = "wengti0608/q-Taxi-v3_note",
            model = model,
            env = env)
```

Show hidden output

Load models downloaded from Hugging Face Hub

- Cannot use built-in **load_from_hub** because this is a custom made model.
- The following code is provided by the tutorial: https://colab.research.google.com/github/huggingface/deep-rl-class/blob/master/notebooks/unit2/unit2.ipynb#scrollTo=AB6n__hhg7YS

```
from huggingface_hub import hf_hub_download

def load_from_hub(repo_id: str, filename: str) -> str:
    """
    Download a model from Hugging Face Hub.
    :param repo_id: id of the model repository from the Hugging Face Hub
    :param filename: name of the model zip file from the repository
    """
    # Get the model from the Hub, download and cache the model on your local disk
    pickle_model = hf_hub_download(
        repo_id=repo_id,
        filename=filename
    )

    with open(pickle_model, 'rb') as f:
      downloaded_model_file = pickle.load(f)

    return downloaded_model_file
```

```
model = load_from_hub(repo_id = "wengti0608/q-Taxi-v3",
                      filename = "q-learning.pkl")

mean_reward, std_reward = evaluate_agent(env = gym.make(model['env_id']),
                                         max_steps = model['max_steps'],
                                         n_eval_episodes = model['n_eval_episodes'],
                                         q_table = model['qtable'],
                                         seed = model['eval_seed'])

print(f"The model has a performance of {mean_reward:.2f} +/- {std_reward:.2f}")
```

q-learning.pkl: 100% 24.6k/24.6k [00:00<00:00, 25.9kB/s]

100% 100/100 [00:00<00:00, 3158.08it/s]

```
The model has a performance of 7.56 +/- 2.71
```