1.a

Please see 'Frozen_Lake_Q_Learning.ipynb' in https://github.com/mattmcm1/ENGR520_HW4 for my code. Code screenshots also in appendix of this doc.

For this part, I used the Frozen Lake environment. I set up the standard 4x4 grid and experimented with both slippery and non-slippery surfaces to see how it affected the training of the agent.

If you run through the whole code, you will see an animation of the guy taking his path for both slippery and non-slippery environments!

1.b

Please see 'Frozen_Lake_PPO.ipynb' in https://github.com/mattmcm1/ENGR520_HW4 for my code. Code screenshots also in appendix of this doc.

For this part, I used the same setup as part 1.a above.

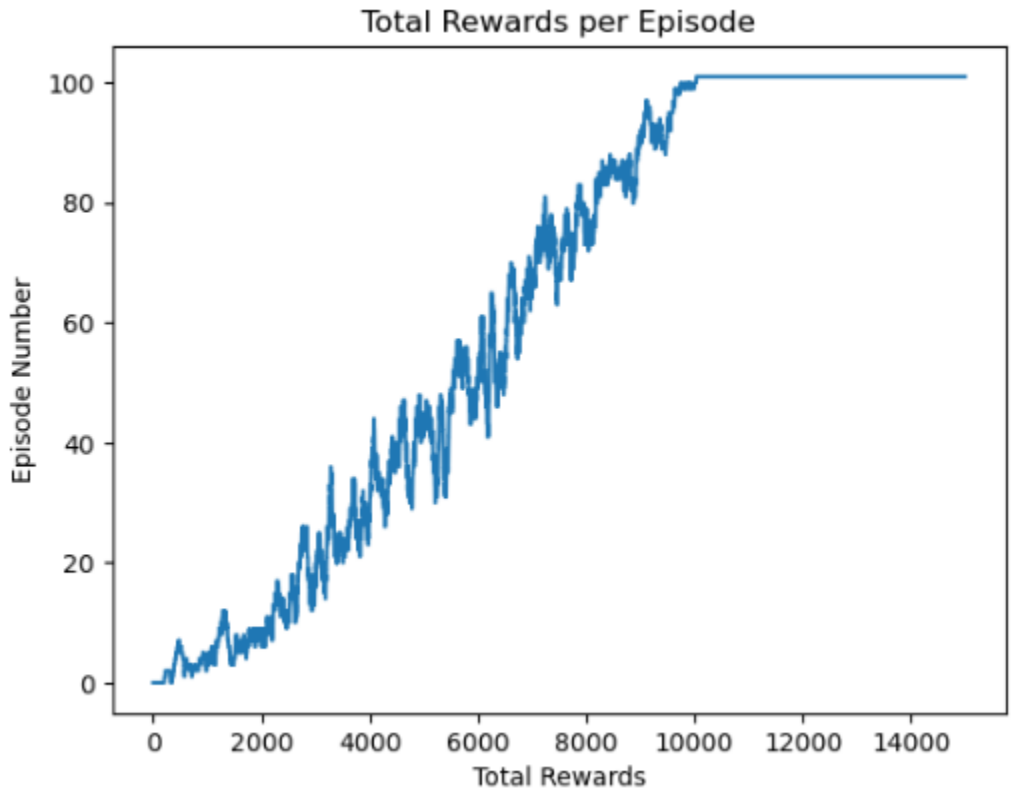
If you run through the whole code, you will see an animation of the guy taking his path for both slippery and non-slippery environments!

2.a - "Notes to your future self".

What I tried

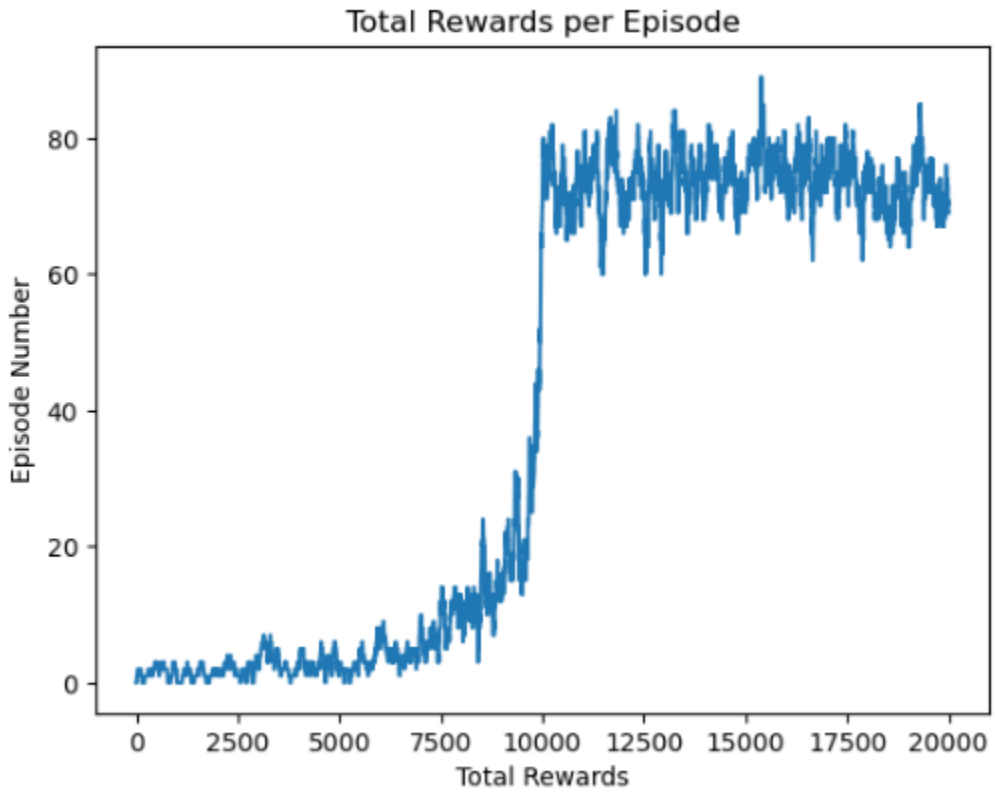
For the Q-learning part:

- To get the codebase up and running, I started by using a simple 4x4 grid with is_slippery set to False. This sets up the environment in a way that there is less randomization, which means that the training will be faster (see documentation here, is_slippery is explained at the bottom). I played around with the number of episodes and the parameters until I got it to work in what seemed like the lowest number of episodes that I could find. It seems to find the path around 10,000 episodes, but I ran 15,000 for a nicer visual. See the non-slip reward plot below:



- Once I got that trained, I moved to a 4x4 grid with is_slippery set to True. I again played around with the number of episodes and parameters until I got it to a 'stable' state. What I mean by this is that it never seems to be able to 100% find the path like the non-slip environment does – this means that sometimes the guy fails and falls into one of the holes. This is due to the randomness of the slippery flag! In the plot below, you will see that the total rewards plateau around 60-80 in most training runs (whereas non-slip

gets around 100). You will also notice that it seems to take about as many episodes (10,000) to reach its plateau, as the non-slip model. See slippery reward plot below:



- In testing the outputs of the final episode (which should theoretically be the correct path), it usually ends up getting there. However, in some cases, the guy does end up falling into one of the holes. The randomness of the slippery flag is probably the reason for this and is why we are not able to get the path 100% right.

For the PPO part:

- I originally wanted to try to replicate this YouTube video by training an agent using a PPO to play Super Mario 64. After doing some research and getting the ROM set up, I realized that it would take way too much time/effort/training resources for a two-week assignment. For this reason, I pivoted back to the Frozen Lake environment since I already had it working from the Q-learning portion.
- I found that the code was easier to write here, as it seems much of the work is already done via stable_baselines3. Since much of the work is done behind the scenes, I made sure to make the PPO call verbose to see at least some outputs. An example of those outputs is shown below:

```
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 7.66        |
|    ep_rew_mean          | 0.86        |
| time/                   |             |
|    fps                  | 1274        |
|    iterations           | 8           |
|    time_elapsed         | 12          |
|    total_timesteps      | 16384       |
| train/                  |             |
|    approx_kl            | 0.017024215 |
|    clip_fraction        | 0.154       |
|    clip_range           | 0.2         |
|    entropy_loss         | -0.838      |
|    explained_variance   | -0.0174     |
|    learning_rate        | 0.0003      |
|    loss                 | 0.0279      |
|    n_updates            | 70          |
|    policy_gradient_loss | -0.0196     |
|    value_loss           | 0.054       |
-----------------------------------------
```

- This is the way that I found easiest to track how the model is training over the number of episodes (total_timesteps) that you give the agent.
- Since I had these outputs, I did not end up plotting the total rewards. You can instead look at the loss values in these tables to track how the agent is improving!

What worked

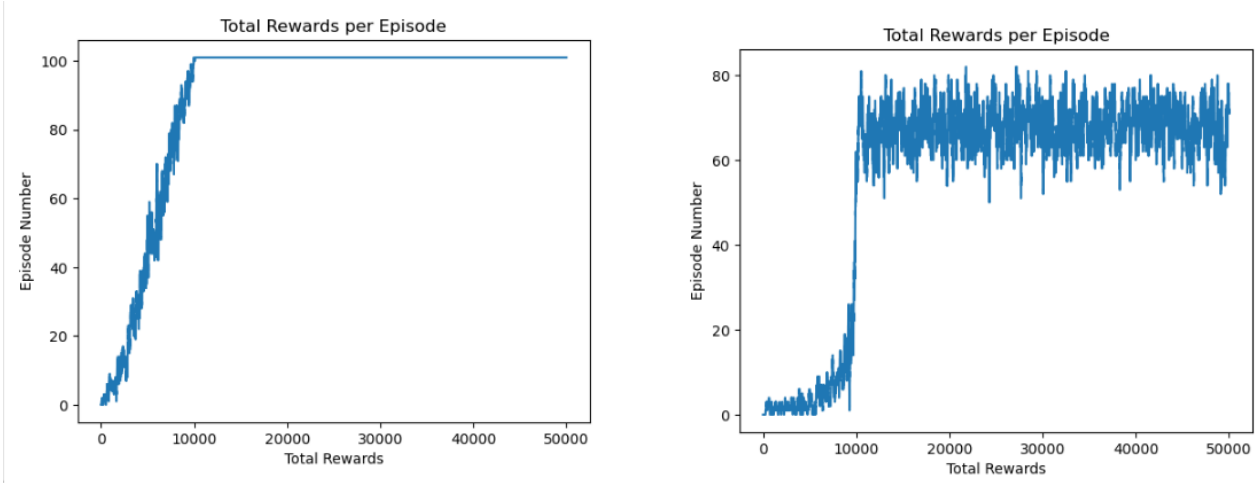
For the Q-learning part:

- After playing around with the parameters, I found that the following set seemed to train the agent well:

```
learning_rate_a = 0.9
discount_factor_g = 0.9 # ~0 = more weight/reward placed on immediate state, ~1: more on future state
epsilon = 1         # how much of the action is random
epsilon_decay_rate = 0.0001
```

- Here is an explanation of each of them and why I chose those values:
    - Learning rate – how quickly the agent updates its Q-values. I chose a high value so that the agent would learn faster. I thought this was appropriate given the simple environment it is seeing.
    - Discount factor – tells the agent how much to weigh future rewards vs current rewards when making decisions. Closer to 0 means it weights current state more, closer to 1 means that it weighs the future state more. I chose a high discount factor here so that the agent sees the bigger picture!

    - Epsilon – determines randomness of an action. A value of 1 signifies completely random. I chose to start with a value of 1 since the agent will not know which way to go otherwise! This will change with the next parameter.
    - Epsilon decay rate – how much to decrease epsilon by in each training episode, so that actions become less random as the agent learns more. I found that this value worked nicely with allowing the agent to see many states in order to figure out a path.
- It also seems like I found a good number of episodes to train – around 10,000 for each. The plots below display what happens when I trained both non-slippery (left) and slippery (right) for 50,000 episodes. You will notice that there is no gain in the non-slip reward plot since it has already found the path, and that the plateau is noticeable in the slip reward plot, as the randomness of the slippery nature prevents the agent from finding the perfect path.



For the PPO part:

- As mentioned, the setup process is much less involved, so there was no parameter setup in my code.
- I did notice a similar pattern to the Q-learning process in that there seems to be a cap on the useful number of episodes!
    - Non-slippery: somewhere around 3,000 episodes, the agent has found the path. Adding more training episodes does nothing to improve the path but ends up taking longer!
    - Slippery: again seems to work somewhere around 10-15,000 episodes. And again seems to gain nothing by training further. The caveat here is that due to the

slippery nature, he does not always find the path! This is just what I observed over my various tests.

What I would try in the future

For the Q-learning part:

- With unlimited time/resources, I would want to see how this performs on a more complex environment. I did some testing on an 8x8 grid, but the small episode training batches that I ran were not very successful. I unfortunately did not have time or resources to train for as long as I could. I would like to revisit this without those constraints.

For the PPO part:

- Once summer break hits, I want to try to get the Super Mario 64 PPO agent up and running! I won't have a time constraint and can leave the training running for as long as it takes. It would be cool if I could get it to work! I may even start with the simpler Mario Bros game first.

References:

- First off, I want to thank this YouTube creator for his extremely helpful frozen lake tutorial. From this, I was able to modify his code to get a great starting point for both parts of the assignment! You will see that my code structure looks similar to his, with changes for this assignment (especially the PPO part)
- I also found the gymnasium documentation to be extremely helpful in setting up my code, as well as the Frozen Lake documentation for the environment I used for all of part 1.
- I found this paper from John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov very useful in the PPO portion of the assignment: Proximal Policy Optimization Algorithms

2.b.i - In your own words, define these terms and explain how they relate to each other: Value, Reward, Quality.

Value: The total expected reward that an agent is able to collect from the current state and onward. This allows the agent to look at the 'big picture' and to look at how promising a certain state is in the long run, instead of just focusing on the immediate available reward.

Reward: Immediate feedback that an agent is given based on the environment. Given immediately after the agent took an action. Basically, tells the agent how good/bad the action that it took was. This is a tool to guide the agent's learning process.

Quality: Also referred to as the Q-value. The Q-value measures the expected total reward of taking an action in a given state and then following a policy after. Basically checks how good of an overall choice a given action is.

How they relate:

- Rewards are the learning signals given to the agent.
- Value summarizes how good each state is, based on future rewards.
- Q-values estimate the value of taking a specific action in a specific state.
- The agent uses the highest Q-values to choose an action, which are then updated based on the immediate rewards, which then get used to estimate values of future states!

2.b.ii - In pseudocode (and supplementary text if needed), define the Q-learning algorithm using a table. Now do the same thing, but for deep Q-learning using a neural network in place of a table.

2.b.ii ¶

Psuedocode for Q-learning algorithm using a table

```
initialize q-table Q with zeros, one for each state/action pair

for _ in episode:
    initialize state
    for step in episode:
        choose action using epsilon-greedy policy from Q[state, :]
        take action, observe reward and next state

        update q-table:
            Q[state, action] = Q[state, action] + alpha * (reward + learning_rate * (max(Q[new state,:]) - Q[state, action])

        state = new state
        break if updated state is the end goal, or a failure point (guy falls in lake)
```

Psuedocode for Q-learning algorithm using a neural network in place of a table

```
initialize replay buffer
initialize Q-network
initialize target network

for _ in episode:
    initialize state
    for step in episode:
        choose action using epsilon-greedy policy from Q-network
        take action, observe reward and next state
        store state, action, reward, next state in replay buffer

        sample random minibatch from replay buffer:
            for state, action, reward, next state from random minibatch:
                target = reward from minibatch + learning rate * max(target(next state))

        do gradient descent on loss: (Q-net(next state, next action) - target)^2

        every N steps, update target network

        state = next state
        break if updated state is the end goal, or a failure point (guy falls in lake)
```

- very similar to first Q-learning algorithm
- replaces Q-table with a neural network that approximates the Q-table
- replay buffer breaks correlation between back-to-back actions

2.b.iii - In pseudocode (and supplementary text if needed), define the PPO algorithm.

2.b.iii

Psuedocode for PPO algorithm

```
initalize policy network and value function

for _ in iteration:
    collect N timestamps of state, action, reward by running policy network in the environment

    compute advantage estimates A_t using GAE (Generalized Advantage Estimation)
    compute returns R_t as sum of discounted rewards for each timestep

    for k epochs:
        for minibatch from data above:
            compute ratio of new policy network to old policy network
            compute surrogate loss
            compute value function loss
            do gradient descent on combined loss:
                L = -surrogate_loss + a * value_func_loss - b * entropy_bonus (a, b are constant weights)
```

Appendix

Frozen Lake Q-Learning

SETUP

```
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
import pickle
```

```
'''
This is a function to train the agent on how to get from start > end without falling into any holes using Q-learning.
How the agent is trained can be changed in the parameters section near the top of the function.

The outputs will be a plot with rewards per episode (basically how well the agent is performing the task),
and a pickle file with all the relevant data during training.
'''
def train(episodes, render=False, slippery=False):
    env = gym.make('FrozenLake-v1', map_name="4x4", is_slippery=slippery, render_mode='human' if render else None) # setup 4x4 la
    q = np.zeros((env.observation_space.n, env.action_space.n)) # init q array

    # init parameters
    learning_rate_a = 0.9
    discount_factor_g = 0.9 # ~0 = more weight/reward placed on immediate state, ~1: more on future state
    epsilon = 1         # how much of the action is random
    epsilon_decay_rate = 0.0001
    rng = np.random.default_rng()

    rewards_per_episode = np.zeros(episodes) # init reward storage

    for i in range(episodes):
        if i % 5000 == 0: # status updates
            print(f"Episode {i}/{episodes}")

        state = env.reset()[0]  # states: 0 to 63, 0=top left corner,63=bottom right corner
        terminated = False      # True when fall in hole or reached goal
        truncated = False       # True when actions > 200

        while(not terminated and not truncated):
            if rng.random() < epsilon: # take random action
                action = env.action_space.sample()
                # actions: 0=left, 1=down, 2=right, 3=up
            else:
                action = np.argmax(q[state,:])

            new_state,reward,terminated,truncated,_ = env.step(action) # do action and collect state, reward, status

            # compute and store q value for state/action pair
            q[state,action] = q[state,action] + learning_rate_a * (
                reward + discount_factor_g * np.max(q[new_state,:]) - q[state,action]
            )

            state = new_state # reset state for loop

        epsilon = max(epsilon - epsilon_decay_rate, 0) # decay epsilon to make movements less rand
```

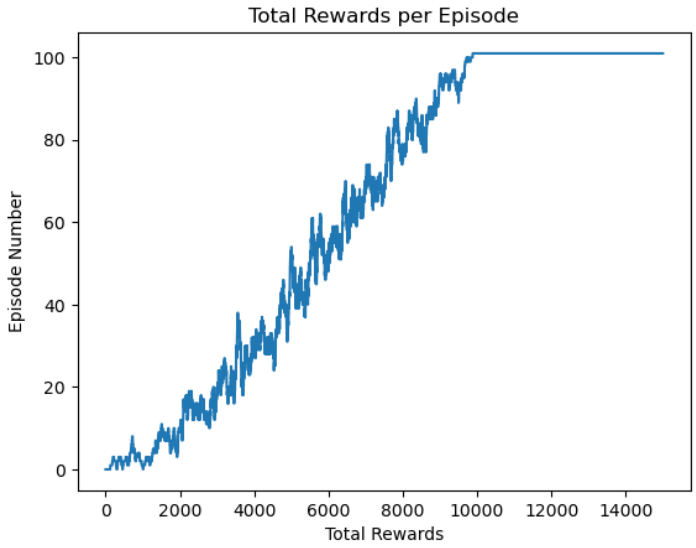
```
        # cleanup
        if(epsilon==0):
            learning_rate_a = 0.0001
        if reward == 1:
            rewards_per_episode[i] = 1

    env.close()

    # collect and plot total rewards per episode
    sum_rewards = np.zeros(episodes)
    for t in range(episodes):
        sum_rewards[t] = np.sum(rewards_per_episode[max(0, t-100):(t+1)])
    plt.plot(sum_rewards)
    plt.title('Total Rewards per Episode')
    plt.xlabel('Total Rewards')
    plt.ylabel('Episode Number')

    # save data in pickle
    f = open("frozen_lake4x4.pkl","wb")
    # f = open("frozen_lake8x8.pkl","wb")
    pickle.dump(q, f)
    f.close()
```

```
'''
This is a near-copy of training, but takes the final episode of the trained agent and display the path in a pop-out window.
Much of the logic is the same but was worth splitting up for cleanliness!
'''
def test(episodes=1, is_training=False, render=False, slippery=False):
    env = gym.make('FrozenLake-v1', map_name="4x4", is_slippery=slippery, render_mode='human' if render else None)

    f = open('frozen_lake4x4.pkl', 'rb')
    # f = open('frozen_lake8x8.pkl', 'rb')
    q = pickle.load(f)
    f.close()

    learning_rate_a = 0.9
    discount_factor_g = 0.9
    epsilon = 1
    epsilon_decay_rate = 0.0001
    rng = np.random.default_rng()

    rewards_per_episode = np.zeros(episodes)

    for i in range(episodes):
        state = env.reset()[0]
        terminated = False
        truncated = False

        while(not terminated and not truncated):
            action = np.argmax(q[state,:])

            new_state,reward,terminated,truncated,_ = env.step(action)

            state = new_state

        epsilon = max(epsilon - epsilon_decay_rate, 0)

        if(epsilon==0):
            learning_rate_a = 0.0001

        if reward == 1:
            rewards_per_episode[i] = 1

    env.close()

    sum_rewards = np.zeros(episodes)
    for t in range(episodes):
        sum_rewards[t] = np.sum(rewards_per_episode[max(0, t-100):(t+1)])
```

Non-Slip

```
# do the training and see rewards
n_episodes = 15000
train(n_episodes, slippery=False)
```

```
Episode 0/15000
Episode 5000/15000
Episode 10000/15000
```



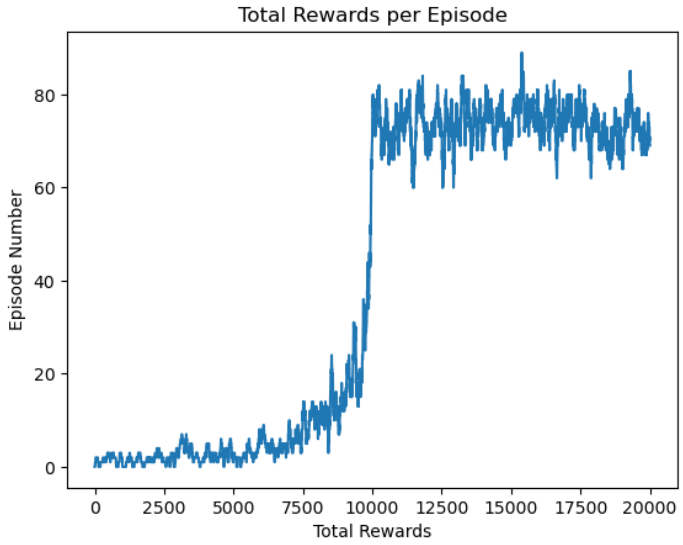

```
# see how the little guy does!
test(render=True, slippery=False)
```

Slippery

Note: acts funky sometimes when run on same kernel as non-slip - would recommend restarting the kernel, running the setup section, and then skipping the non-slip section and coming directly here. Results will be better!

```
# do the training and see rewards
n_episodes = 20000
train(n_episodes, slippery=True)
```

```
Episode 0/20000
Episode 5000/20000
Episode 10000/20000
Episode 15000/20000
```




```
# see how the little guy does!
test(render=True, slippery=True)
```

Frozen Lake PPO

SETUP

```
import gymnasium as gym
import matplotlib.pyplot as plt
import numpy as np
import os
import torch

from stable_baselines3 import PPO
from stable_baselines3.common.env_util import make_vec_env

MODEL_PATH = "ppo_frozenlake.zip"
```

```
'''
Train an agent on the FrozenLake 4x4 environment using PPO (Proximal Policy Optimization).
Saves the trained model to disk.
'''
def train(episodes=15000, slippery=False):
    # setup environment
    env = make_vec_env(
        lambda: gym.make("FrozenLake-v1", map_name="4x4", is_slippery=slippery),
        n_envs=1
    )

    # setup, train, save model
    model = PPO("MlpPolicy", env, verbose=1, device='cpu') # apparently using the cpu is better here, so force it
    model.learn(total_timesteps=episodes)
    model.save("ppo_frozenlake_model")

    env.close()
```

```
'''
Test a trained PPO agent on the FrozenLake 4x4 environment.
Displays the environment if render=True.
'''
def test(episodes=1, render=False, slippery=False):
    # load model
    model = PPO.load("ppo_frozenlake_model")

    # setup environment
    env = gym.make(
        "FrozenLake-v1",
        map_name="4x4",
        is_slippery=slippery,
        render_mode='human' if render else None
    )


    rewards_per_episode = np.zeros(episodes)

    for i in range(episodes):
        obs, _ = env.reset()
        total_reward = 0
        done = False

        while not done:
            action, _ = model.predict(obs, deterministic=True)
            obs, reward, terminated, truncated, _ = env.step(int(action))
            total_reward += reward
            done = terminated or truncated

        rewards_per_episode[i] = total_reward

    env.close()
```

Non-slippery

```
# train the model and output stats
train(episodes=3000, slippery=False)
```

```
# see how the guy does!
test(episodes=1, render=True, slippery=False)
```

Slippery

```
train(episodes=15000, slippery=True)
```

```
test(episodes=1, render=True, slippery=True)
```