Computer Vision and Robotics Final Project

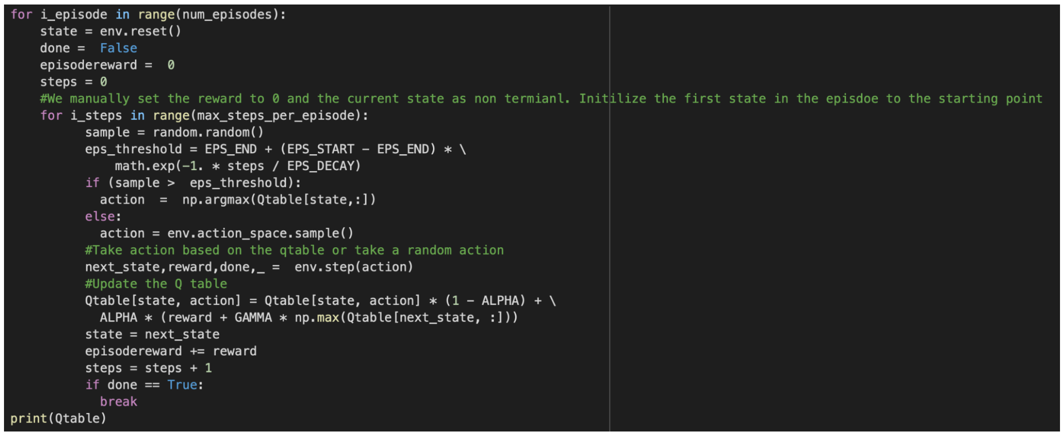
Brianna Solano Aguilar, Felix Shames

Intro

For the Final Project we took on the topic of Maze Navigation Using Reinforcement Learning. We trained a model to solve The Frozen Lake problem which is a slightly different 4x4 maze than basic ones. The maze is depicted by the letters on the grid, there is only 1 S spot and that is where the agent starts the maze. Then there is also 1 G on the grid, which is the goal. The other letters on the maze are F for the Frozen Lake where the floor could either be normal or the floor could be slippery. If the floor is slippery it has a chance to make the agent take a randomized action, instead of their chosen action, which includes the chance to do nothing. This status of the lake will make it harder for the agent to win and to train consistently. Through the changes to the floor type it changes the problem from a stochastic solution where the consequences of actions are always the same, to a deterministic one where the outcome is more unknown but the agent can still make the best choice based on probability and its current location in the maze. Finally, unlike other mazes, it has holes instead of walls, which are shown by the letter H. If the agent lands on a hole, then they instantly fail that run and start over at the beginning of the maze, this creates the need for the agent to pick the best option in their situation so they can minimize the risk of themselves running into a hole.

Methods

There were two main methods used during the training of this project using the frozen lake environment in Open Gym AI. We use a model-less algorithm such as Q table and we also use value iteration. Before implementing any of the methods we had to declare a variety of important values such as the discount rate, the learning rate, exploration rate, and exploration rate of decay. We also had to initialize our environment with our given parameters. We initialized the Frozen Lake environment to not have a slippery floor. We finally have to create two empty arrays such that we can record the steps and total reward obtained in every iteration of an episode. Once everything is declared we implement Q-table to find the shortest path in the frozen lake environment while also avoiding the holes that are placed on the map. We first start with an array of size m by n zeros where m is the total number of states in the environment (16) and n is the total number of actions that the agent can take (4). We can finally implement our algorithm, as shown in figure 1, to update our Q-table with the appropriate Q-values.



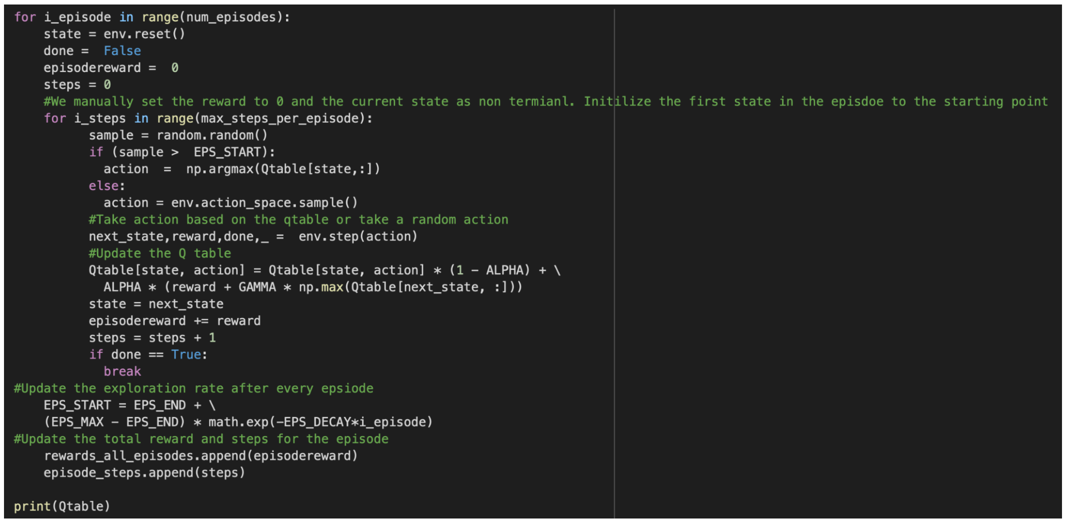
```
for i_episode in range(num_episodes):
    state = env.reset()
    done =  False
    episodereward =  0
    steps = 0
    #We manually set the reward to 0 and the current state as non termianl. Initilize the first state in the episdoe to the starting point
    for i_steps in range(max_steps_per_episode):
          sample = random.random()
          eps_threshold = EPS_END + (EPS_START - EPS_END) * \
              math.exp(-1. * steps / EPS_DECAY)
          if (sample >  eps_threshold):
            action  =  np.argmax(Qtable[state,:])
          else:
            action = env.action_space.sample()
          #Take action based on the qtable or take a random action
          next_state,reward,done,_ =  env.step(action)
          #Update the Q table
          Qtable[state, action] = Qtable[state, action] * (1 - ALPHA) + \
            ALPHA * (reward + GAMMA * np.max(Qtable[next_state, :]))
          state = next_state
          episodereward += reward
          steps = steps + 1
          if done == True:
            break
print(Qtable)
```

Figure 1: Q-table algorithm using the Epsilon-Greedy Exploration Strategy

Before getting into the code, there needs to be an explanation of the strategy used to obtain our Q-table. We implemented the Epsilon-Greedy Exploration Strategy. As mentioned before the Q-table is composed of all zeros, which signifies that our agent doesn't know anything about the environment. Our Q-table consists of estimated optimal future values to an action-state pair. Since nothing is known about the world when we have to explore our environment and as we explore we update our Q-table. This phase is known as exploration and is simply a period of trial and error. Once we have a better understanding of our environment, we can move onto the next phase which is exploitation. At the end of the day we want to take an action at a state that leads to the highest expected reward. Our agent traversed through the states by taking the best action in which it expects the maximum expected reward.

As you see in the code we iterate through a loop that ranges from 0 to the total number of episodes, which in this case is 10,000. Each episode resets the environment, sets done to false, and episodereward and steps to 0. The done variable just signifies that the agent didn't step into a state that terminates the program. We also set the maximum amount of steps the agent can take to 100. This ensures that the episodes don't continuously loop for a long time. Within the episode we loop through the number of steps taken which range from 0 to 100. For every step taken we randomly select a number. That number will help us determine whether we conduct exploration or exploitation. If the random value that we obtain is less than epsilon, then we take a random action in the 4 available actions in the environment. Otherwise, we conduct exploitation and take the best action in which we expect the maximum expected reward. Once we determine the chosen action, we take the action in the environment. When an action is taken in the environment we are returned 4 objects, new state (location on the grid), reward, done (If the new state is terminal), and extra information. The Q-table is then updated and the new state becomes the current state. If the state doesn't terminate the program, a new action is taken, otherwise the program terminates and we enter into a new episode. It is noted that our epsilon, which helps determine if we conduct exploration or exploitation, is also affected by the amount of steps taken in their respective episode.

We perform another Epsilon Greedy Exploration strategy, using the same Q table implemented before but rather than epsilon being affected by the total number of steps, it's affected by the total number of episodes. We also set new values for discount rate, the learning rate, exploration rate, and exploration rate of decay. This finally allows us to obtain our final optimized Q-table. Results can be seen in Graph 1 below.



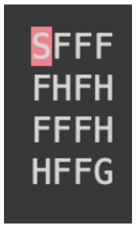
```
for i_episode in range(num_episodes):
    state = env.reset()
    done =  False
    episodereward =  0
    steps = 0
    #We manually set the reward to 0 and the current state as non termianl. Initilize the first state in the episdoe to the starting point
    for i_steps in range(max_steps_per_episode):
          sample = random.random()
          if (sample >  EPS_START):
            action  =  np.argmax(Qtable[state,:])
          else:
            action = env.action_space.sample()
          #Take action based on the qtable or take a random action
          next_state,reward,done,_ =  env.step(action)
          #Update the Q table
          Qtable[state, action] = Qtable[state, action] * (1 - ALPHA) + \
            ALPHA * (reward + GAMMA * np.max(Qtable[next_state, :]))
          state = next_state
          episodereward += reward
          steps = steps + 1
          if done == True:
            break
#Update the exploration rate after every epsiode
    EPS_START = EPS_END + \
    (EPS_MAX - EPS_END) * math.exp(-EPS_DECAY*i_episode)
#Update the total reward and steps for the episode
    rewards_all_episodes.append(episodereward)
    episode_steps.append(steps)

print(Qtable)
```

Figure 2: New Q-table algorithm using the Epsilon-Greedy Exploration Strategy

This method is also conducted in the same environment again, but using a slippery floor instead. This means that actions taken don't guarantee that the agent will move in that direction. For example, let's say we are in state 0 and force the agent to take a 'Right' action. We expect the agent to move to state 1, but because the floor is slippery the agent doesn't move or moves to state 4. We first use the already developed Q-table and update it to a slippery frozen lake environment. Since the grid is exactly the same, we expect the agent to do exceptionally well when compared to starting with a blank Q-table. Results for this experiment can be seen in Graph 2. We conduct the experiment using a blank Q-table and these results can be seen in Graph 3.

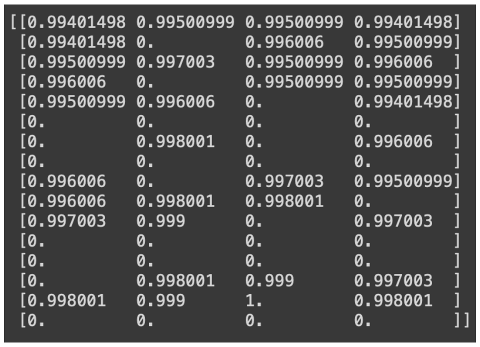
We lastly use the value iteration to extract the policy. This algorithm is done using the Frozen Lake environment with a non-slippery floor. This is done by calculating the values of the state-action pair for all possible actions in that particular state. Once the value for the difference between the value for the new state and the value for the old state are relatively small, we can terminate the iteration. This process helps us obtain the value function and from there we are able to extract the policy. From the value function we simply calculate the state-action value for all possible actions. From there we select the action that leads to the highest state-action value.

Results



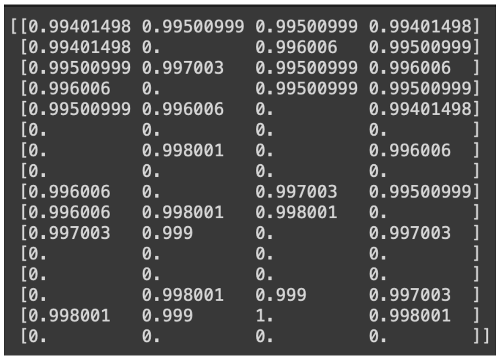
```
SFFF
FHFH
FFFH
HFFG
```

Figure 3: Environment



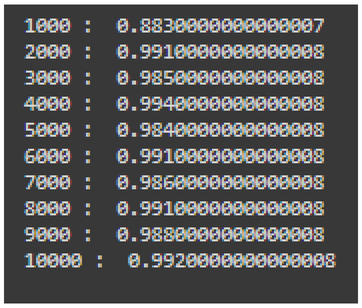
```
[[0.99401498 0.99500999 0.99500999 0.99401498]
 [0.99401498 0.         0.996006   0.99500999]
 [0.99500999 0.997003   0.99500999 0.996006  ]
 [0.996006   0.         0.99500999 0.99500999]
 [0.99500999 0.996006   0.         0.99401498]
 [0.         0.         0.         0.        ]
 [0.         0.998001   0.         0.996006  ]
 [0.         0.         0.         0.        ]
 [0.996006   0.         0.997003   0.99500999]
 [0.996006   0.998001   0.998001   0.        ]
 [0.997003   0.999      0.         0.997003  ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.998001   0.999      0.997003  ]
 [0.998001   0.999      1.         0.998001  ]
 [0.         0.         0.         0.        ]]
```

Table 1: First version of the Q-table using Epsilon-Greedy Exploration Strategy



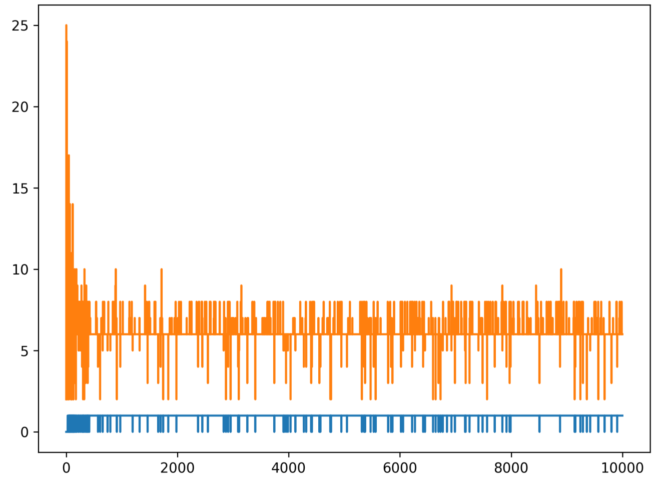
```
[[0.99401498 0.99500999 0.99500999 0.99401498]
 [0.99401498 0.         0.996006   0.99500999]
 [0.99500999 0.997003   0.99500999 0.996006  ]
 [0.996006   0.         0.99500999 0.99500999]
 [0.99500999 0.996006   0.         0.99401498]
 [0.         0.         0.         0.        ]
 [0.         0.998001   0.         0.996006  ]
 [0.         0.         0.         0.        ]
 [0.996006   0.         0.997003   0.99500999]
 [0.996006   0.998001   0.998001   0.        ]
 [0.997003   0.999      0.         0.997003  ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.998001   0.999      0.997003  ]
 [0.998001   0.999      1.         0.998001  ]
 [0.         0.         0.         0.        ]]
```

Table 2: First version of the Q-table using Epsilon-Greedy Exploration Strategy

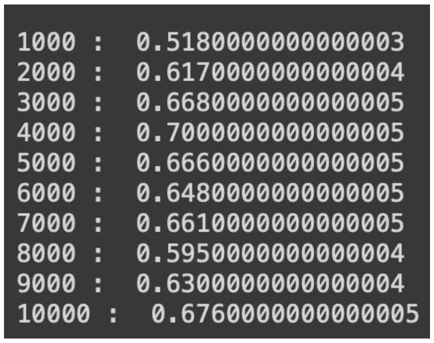


```
1000 :  0.8830000000000007
2000 :  0.9910000000000008
3000 :  0.9850000000000008
4000 :  0.9940000000000008
5000 :  0.9840000000000008
6000 :  0.9910000000000008
7000 :  0.9860000000000008
8000 :  0.9910000000000008
9000 :  0.9880000000000008
10000 :  0.9920000000000008
```

Table 3: Average reward for every 1000 episodes using a non slippery floor

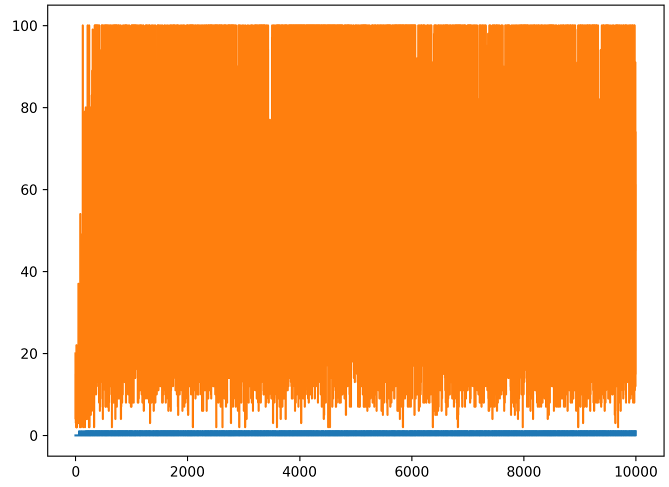


Graph 1: Results of Deep Learning using Q-table using a non slippery floor

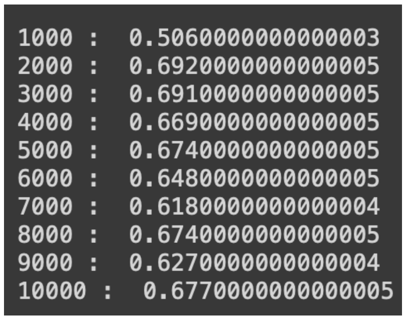


```
1000 :  0.5180000000000003
2000 :  0.6170000000000004
3000 :  0.6680000000000005
4000 :  0.7000000000000005
5000 :  0.6660000000000005
6000 :  0.6480000000000005
7000 :  0.6610000000000005
8000 :  0.5950000000000004
9000 :  0.6300000000000004
10000 :  0.6760000000000005
```

Table 4: Average reward for every 1000 episodes using a slippery floor and pre-made Q-table from first model

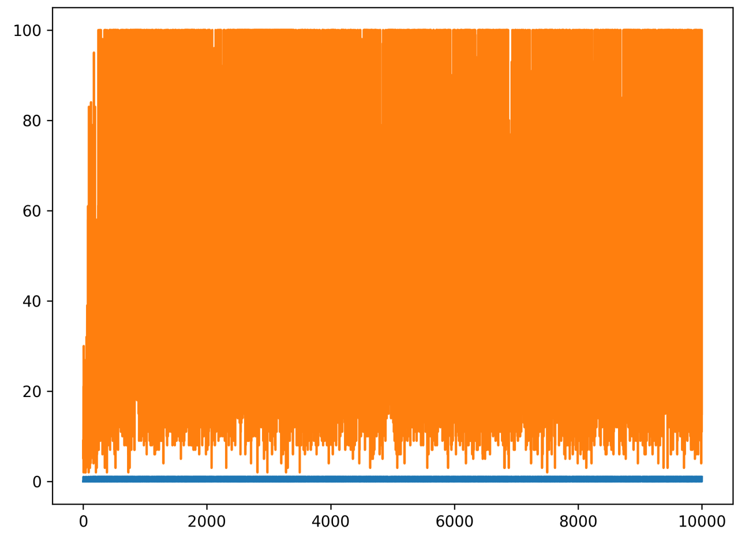


Graph 2: Results of Deep Learning using Q-table using a slippery floor and pre-made Q-table from first model

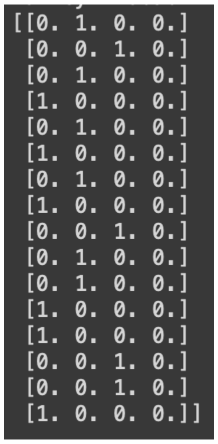


```
1000 :  0.5060000000000003
2000 :  0.6920000000000005
3000 :  0.6910000000000005
4000 :  0.6690000000000005
5000 :  0.6740000000000005
6000 :  0.6480000000000005
7000 :  0.6180000000000004
8000 :  0.6740000000000005
9000 :  0.6270000000000004
10000 :  0.6770000000000005
```

Table 5: Average reward for every 1000 episodes using Q-table using a slippery floor

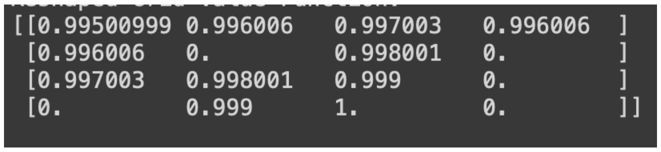


Graph 3: Results of DeepLearning using Q-table using a slippery floor



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

Table 6: Policy Probability distribution



```
[[0.99500999 0.996006   0.997003   0.996006  ]
 [0.996006   0.         0.998001   0.        ]
 [0.997003   0.998001   0.999      0.        ]
 [0.         0.999      1.         0.        ]]
```

Table 7: Value Function

Conclusion

From the results of the first graph it can be seen that our first model that was trained with a normal floor was able to quickly reduce the amount of steps it took to get to the goal. From graph 2 it can be seen that the pretrained model did not fare well on slippery floors, even in the same exact maze it had already solved. The agent hit the max number of steps almost every time, with only a few victories. Even though the model already knew the shortest path, the fact that it was trained didn't help it at all since the randomness of the slippery floor made it constantly have to readjust and relearn to solve the maze. However, since it was being affected by the randomness of the floor almost at all times, it could never really choose a path to get to the goal. Unfortunately for the agent, solving the maze had no effect on the rest of the future episodes. The results from graph 3 are almost the exact same as graph 2, which was an untrained model v.s. A trained model. From these results we can conclude that whether the model is trained on the maze or not, their performance will not improve on the slippery floor maze.