Computer Vision and Robotics Final Project

Brianna Solano Aguilar, Felix Shames

Intro

- For the project we chose the topic of Maze Navigation Using Reinforcement Learning
  - Instead of solving a normal maze we solved The Frozen Lake Problem
    - The difference is that instead of walls the frozen lake has holes, which instantly fail the agent and make them start over
    - Another big difference between the two is that the floor of the lake can either be slippery or not slippery
      - If it's not slippery than the maze functions like normal and the agent can easily get to the goal
      - But if it is slippery then on any move the agent makes there's a chance for it to get a random outcome
      - Turning the floor from normal to slippery completely changes the type of problem from the original stochastic solution, where its the same every time to a deterministic solution where the agent must make the best choice based on probability
    - These factors make it much more difficult for the agent to complete the maze and almost impossible to train the agent to continually complete the maze due to the randomness

Method

- Two methods were used during this project to obtain the shortest path to compete the game
  - Q-tables
  - Value Iteration Function
- Before implementing any any of our methods we first had to declare certain variables
  - Frozen Lake Environment - Slippery or Non Slippery floor
  - Discount Rate
  - Learning Rate
  - Exploration Rate
  - Exploration decay
  - Two arrays to stored data
    - An array to store total reward per episode
    - An array to store total steps per episode

Q-tables

- Our Goal - Find the shortest path in the frozen lake environment to the objective while also avoiding the holes that are placed on the map
- To implement we first have to initialize our Q-table
  - M by N zero array
    - M → Total States → 16
      - Its a 4 by 4 grid with 15 locations
    - N → Total Actions → 4
      - Agent can go left,right,up,down
- We can implement our algorithm to update the Q-table with the appropriate Q-values

Epsilon-Greedy Exploration Strategy

- Q-table consisted of estimated optimal future values of an action-state pair
- Our Q-table as of now is only composed of zeros
  - Our agent doesn't know anything about the world
- We have to explore our environment
  - This phase is known as exploration
- Once we know more about the world we take actions that lead the highest reward
  - This phase is known as exploitation

Code

```
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.
    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)
```

- We iterate through 10,000 episodes
- We set a maximum amount of steps taken as 100
    - This stops an episodes from running for too long
- For every step taken we randomly select a number
    - This helps us determine whether we conduct exploration or exploitation
    - If that random number is smaller than epsilon exploration rate, then we select a random action
    - Otherwise, we select the best action
- We take the action in the environment and are returned 4 variables
    - New state (location on the grid)
    - Reward
    - Done (Returns true if the new state is terminal)
    - Extra information
- The Q-table is updates and the new state becomes the current state
- If the state doesn't terminate the program, a new action is take, otherwise the program terminates and a new episodes starts
- Our Exploration rate is affected by amount of steps in their respective episode

Second Q-table

- A second Epsilon Greedy Exploration strategy, using the same Q-table implemented before.
    - But rather then the exploration rate being affected by total steps it was affected by total number of episodes elapsed
- New values for such as discount rate, the learning rate, exploration rate, and exploration rate of decay were given
- From there we obtained our final Q-table

```
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. In
    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)
```

Slippery Floor

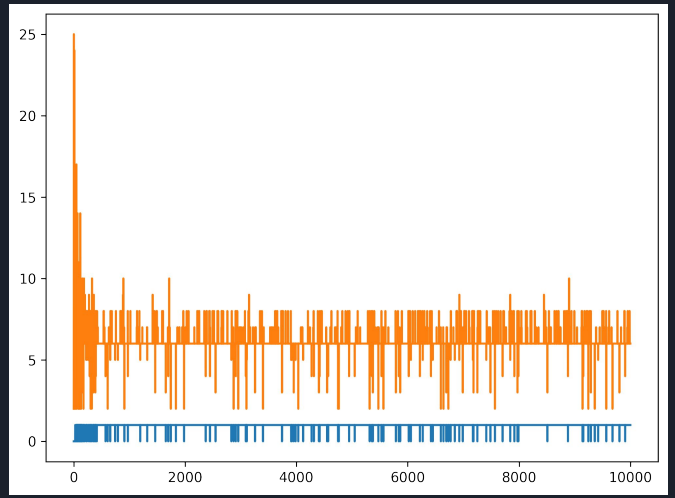
- We conducted the same method in the same environment again, but using a slippery floor instead
  - This didn't guarantee that the action that the agent would take would be definite
- We first used the already developed Q-table and update it to a slippery frozen lake environment
- We then conducted the method using a blank Q-table

Value Iteration Function

- We used the Frozen Lake environment with a non slippery floor
- We calculate the values of the state-action pair for all possible action in that particular state
- Once the value for the different between the value for the new state and the value for the old state are relatively small, we 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 actions that leads to the highest state- action value

First Model Results

Results of Deep Learning with Q-table using a non slippery floor

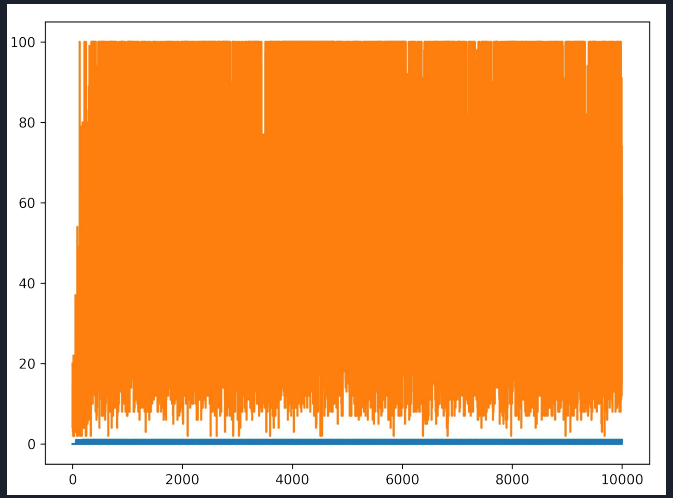


Average reward for every 1000 episodes using a non slippery floor

```
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
```

Second Model Results

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

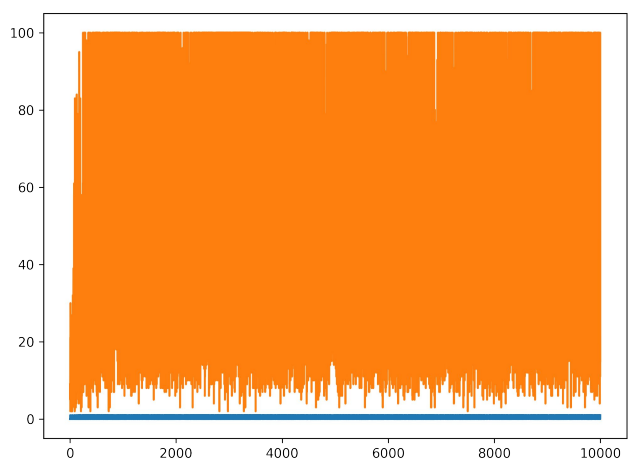


Average reward for every 1000 episodes using a slippery floor and pre-made Q-table from first model

```
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
```

Third Model Results

Results of Deep Learning with Q-table using a slippery floor



Average reward for every 1000 episodes using a slippery floor

```
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
```

Policy

```
Reshaped Grid Policy (0=left, 1=down, 2=right, 3=up):
[[1 2 1 0]
 [1 0 1 0]
 [2 1 1 0]
 [0 2 2 0]]
```

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
- The results from graph 3 are almost the exact same as graph 2
  - 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