Student No. 756011
Zeinab R. Yousefi

Assignment 3

ELEC-E8125
October 25, 2019

1 Task 1 Implement Q-learning for the Cartpole environment

Discretizing Cartpole states to a finite grid is shown in algorithm below 1:

Algorithm 1: State dscritizing algorithm

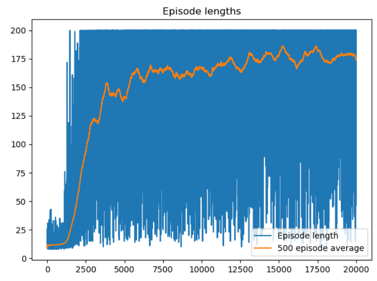
```
def get_discrete_state(state):
    discrete_state = np.zeros(len(state))
    discrete_state[0] = np.argmin(np.abs(x_grid - state[0]))
    discrete_state[1] = np.argmin(np.abs(v_grid - state[1]))
    discrete_state[2] = np.argmin(np.abs(th_grid - state[2]))
    discrete_state[3] = np.argmin(np.abs(av_grid - state[3]))
    return tuple(discrete_state.astype(np.int))
```

The q-table is filled using the following algorithm 2:

Algorithm 2: Qlearning algorithm

```
  q_grid = np.zeros((discr, discr, discr, discr, num_of_actions)) + initial_q

# Training loop
ep_lengths, epl_avg = [], []
for ep in range(episodes+test_episodes):
    test = ep > episodes
    state, done, steps = env.reset(), False, 0
    epsilon = a / (a + ep) # 0.2    # T1: GLIE/constant, T3: Set to 0

    while not done:
        discrete_state = get_discrete_state(state)
        # TODO: IMPLEMENT HERE EPSILON-GREEDY
        #action = int(np.random.rand()*2)
        #action = np.argmax(q_grid[discrete_state])

        action_probabilities = epsilonGreedy(q_grid, epsilon,
                                num_of_actions, discrete_state)

        # choose action according to the probability distribution
        action = np.random.choice(np.arange(
            len(action_probabilities)),
            p=action_probabilities)

        new_state, reward, done, _ = env.step(action)
        new_discrete_state = get_discrete_state(new_state)

        if not test:
            # TODO: ADD HERE YOUR Q_VALUE FUNCTION UPDATE
```

[ϵ = 0.2]



[ϵ decreasing]

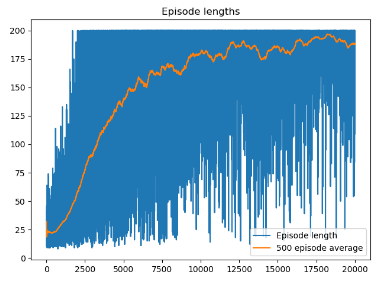


Figure 1: Training performance

```
            max_future_q = np.max(q_grid[new_discrete_state])
            current_q = q_grid[discrete_state + (action, )]
            new_q = (1 - alpha) * current_q + alpha * (reward + gamma * max_future_q)
            q_grid[discrete_state + (action, )] = new_q

            if (done == True):
                current_q = q_grid[discrete_state + (action,)]
                new_q = (1 - alpha) * current_q + alpha * (gamma * current_q)
                q_grid[discrete_state + (action,)] = new_q

            pass
        else:
            env.render()
        state = new_state
        steps += 1
```

The q values for ϵ = 0.2 is saved in "q_values_02Fixed.npy" and for decreasing ϵ from 1 to 0.1 is saved in "q_values_decreasing.npy".

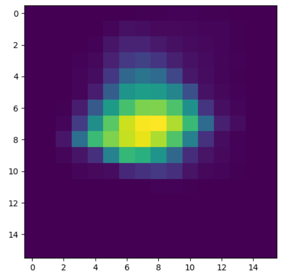
The reward plots for ϵ = 0.2 and ϵ decreasing is shown in the figure **??**.

2 Task 2: Use your Q-function values to calculate the optimal value function of each state

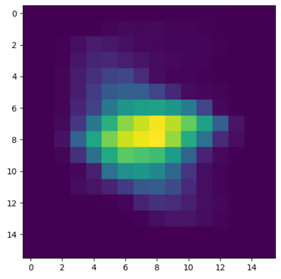
The implementation of value function and heatmap is shown in the algorithm **??**. The state values and policy are saved in numpy array "values" and "policy" (attached to the submission files).

Algorithm 3: Value functions using the qlearning and heat map

```
    # Calculate the value function
    values = np.zeros(q_grid.shape[:-1])  # TODO: COMPUTE THE VALUE FUNCTION FROM THE Q-GRI
    values = np.sum(q_grid, axis=-1)
```



[$\epsilon = 0.2$]



[ϵ decreasing]

Figure 2: Heatmap

```
#(1 - target_eps) * np.max(q_grid, axis=4)  + (target_eps/num_of_actions) * np.sum(q_gr

np.save("value_func_eps_decreasing.npy", values)  # TODO: SUBMIT THIS VALUE_FUNC.NPY AN


# Plot the heatmap
# TODO: Plot the heatmap here using Seaborn or Matplotlib
heat = np.zeros([16, 16])
for i in range(16):
    for j in range(16):
        heat += values[:, i, :, j]
heat = heat / (16*16)

plt.imshow(heat)
plt.show()
```

Heatmaps for the $\epsilon = 0.2$ and decreasing ϵ are shown in the figure 2.

Question 1 What do you think the heatmap would have looked like:

- before the training?

  Before any training, the heatmap is an empty plot. Since we have initialized q-grid with zero's and none of the states are touched.

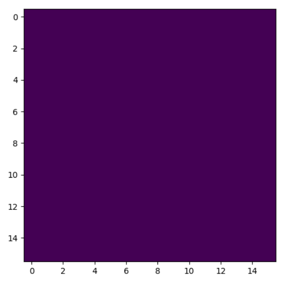
- after a single episode?

  After training the q-grid only for 1 episode, the heatmap looks like Figure 3. The agent is mostly exploring around the initial state which is at (7,7,7,7) and therefore the value functions on the other areas are still zero.

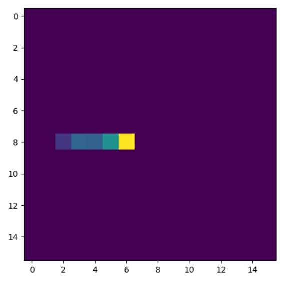
- halfway through the training?

  It can be seen that more and more episodes are being explored. After 10000 episodes, agent goes further than initial states and compared to the heatmap after 20000 episodes, it seems that the value function has started to converge.

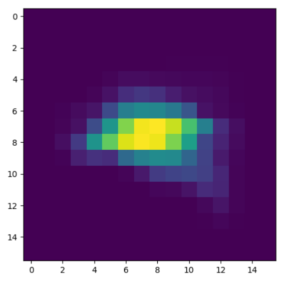
Task 3 Set 2 to zero, effectively making the policy greedy w.r.t. current Q-value estimates.



[Before training]

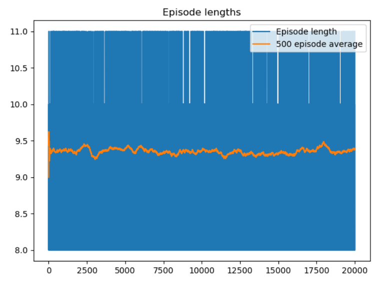


[After a single episode]

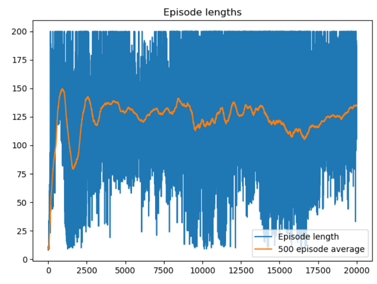


[halfway through training]

Figure 3: Heatmap at different stages of training

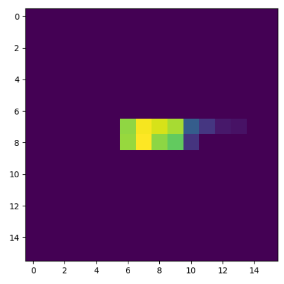


[ini_q = 0]

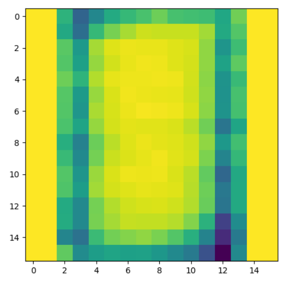


[ini_q = 50]

Figure 4: Training performance comparison



[ini_q = 0]



[ini_q = 50]

Figure 5: Heatmap comparison

- Question 2.1 – In which case does the model perform better?

  The model that its q has been initialized with 50 perform better. The figures related to the training performances and heatmaps for these two different cases are shown in the figure 4 and 5 respectively. It can be seen that when the initial q is zero, the reward hardly reaches to 9.5, however when the q is initialized at 50, the reward reches to 150.

- Question 2.2 – Why is this the case? This optimistically initialization allows the agent to explore the environment more and more.

- Question 2.3 – How does the initialization of Q values affect exploration?

  According to the Sutton and Barto reference book : "Initial action values can also be used as a simple way to encourage exploration." When the $\epsilon = 0$ and initial q is also 0, it means that the agent is acting greedy, and does not explore at all. But when we initialize the q-value optimistically, although it may works worse initially, because it is exploring, but eventually by decreasing the exploration it will perform better.

3 Task 4 – Modify your code for Task 1 and try to apply it in the Lunar lander environment.

The parts of code that has been modified to be implemented in the land lunar environment is shown in algorithm 4.

Algorithm 4: modification of the above algorithm for Land Lunar environment

```
# Reasonable values for Land Lunar discretization
discr = 16
bnr = 2

# For LunarLander, use the following values:
#         [  x     y   xdot  ydot  theta  thetadot  cl   cr
s_min = [  -1.2,  -0.3,  -2.4,   -2,   -6.28,   -8,    0,    0  ]
s_max = [   1.2,   1.2,   2.4,    2,    6.28,    8,    1,    1  ]

def create_discrete_grid(s_min, s_max, discr, bnr):
    state_grid = []
    for i in range(len(s_min) - 2):
        state_grid.append( [np.linspace(s_min[i], s_max[i], discr)])
    state_grid.append([np.linspace(s_min[6], s_max[6], bnr)])
    state_grid.append([np.linspace(s_min[7], s_max[7], bnr)])
    return state_grid


    # Parameters
gamma = 0.98
alpha = 0.1
target_eps = 0.1
a = np.int((episodes * target_eps) / (1-target_eps))  # TODO: Set the correct value.
initial_q = 0  # T3: Set to 50

state_grid = create_discrete_grid(s_min, s_max, discr, bnr)
def get_discrete_state(state):
    discrete_state = np.zeros(len(state))
    for i in range(len(state) - 2):
        discrete_state[i] = np.argmin(np.abs(state_grid[i] - state[i]))
    discrete_state[6] = state[6]
    discrete_state[7] = state[7]
    return tuple(discrete_state.astype(np.int))

q_grid = np.zeros((discr, discr, discr, discr, discr, discr,
                   bnr, bnr, num_of_actions)) + initial_q
```

- Question 3.1 – Is the lander able to learn any useful behaviour?

Lander is not able to learn completely land on the ground between two flag poles. In half of the test time it was landing outside of the flag poles. The training performance is shown in Figure 6

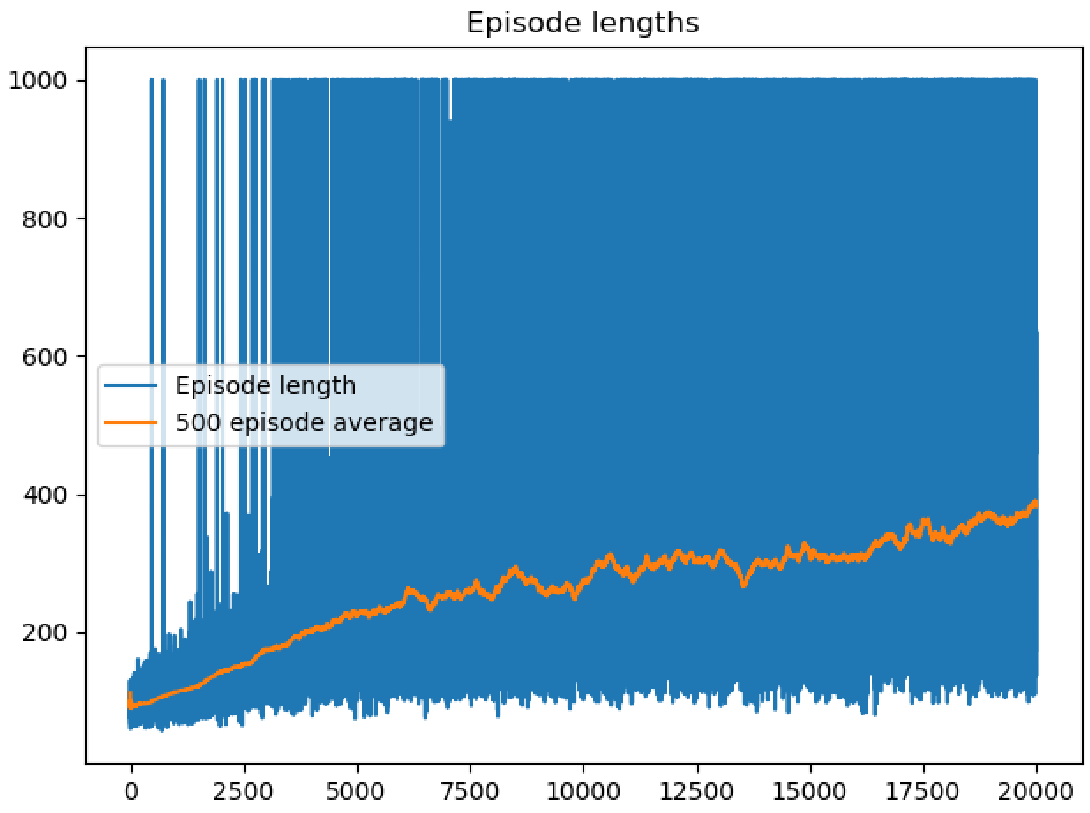


Figure 6: Training performance of q_learning algorithm in land lunar problem

- Question 3.2 – Why/why not?

  I think the main issue is related to the discretization and the small sample efficiency. If we could consider the state space as continuous space it may perform better. This can be done using function approximation.