Exercise 3

Eugeniu Vezeteu - 886240
ELEC-E8125 - Reinforcement Learning

October 7, 2020

1 Task 1

In this task Q-learning algorithm was implemented.
The following code implements the epsilon-greedy policy.

```
def get_action(state, q_values, greedy=False):
    state = get_cell_index(state) #convert state from
       continuous to discrete
    # TODO: Implement epsilon-greedy
    if greedy or np.random.random() >= epsilon:
        #greedy is true -> we test the agent, or prob 1-epsilon
        action = np.argmax(q_values[state])
    else: #random action with prob epsilon
        action = int(np.random.random() * num_of_actions)

    return action
```

The next code implements the Q table update

```
def update_q_value(old_state, action, new_state, reward, done,
    q_array, steps, on_policy=True):
     # TODO: Implement Q-value update
     s = get_cell_index(old_state)   #current state
     s_ = get_cell_index(new_state)  #next state

     if done and steps < 199: #Q(terminal,:) is 0
         q_grid[s][:] = 0.
     else: #Q update formula
         if on_policy:
             a_ = get_action(new_state, q_grid, greedy=True)
             target = reward + (gamma * q_grid[s_][a_]) - q_grid
                [s][action]
         else: #off_policy
             target = reward + (gamma * max(q_grid[s_])) -
                q_grid[s][action]

         q_grid[s][action] += alpha * target
```

Figure 1 show the result of training for 20000 episodes using epsilon-greedy method, with constant $\epsilon = 0.2$.

According to lecture notes, the Greedy in limit with infinite exploration (GLIE) formula is: $\epsilon = \frac{a}{a+k}$.
So, in our case, desired ϵ is 0.1 and k = 20000.
The value of the constant a was computed with the below formula

$$\epsilon = \frac{a}{a+k} => a = \epsilon * a + \epsilon * k => a - \epsilon * a = \epsilon * k => a * (1-\epsilon) = \epsilon * k => a = \frac{\epsilon * k}{1-\epsilon} = 199$$

Figure 2 show the result of training for 20000 episodes using GLIE method, with desired $\epsilon = 0.1$. after 20000 episodes

Files *q_values_task1.npy* and *value_func_task1* contains the Q-value and Value function for CartPole with GLIE.

2 Task 2

On this task the optimal value function was computed from Q table. The heatmap was plotted in terms of x and θ, and averaging the values over $\dot{x}$ and $\dot{\theta}$

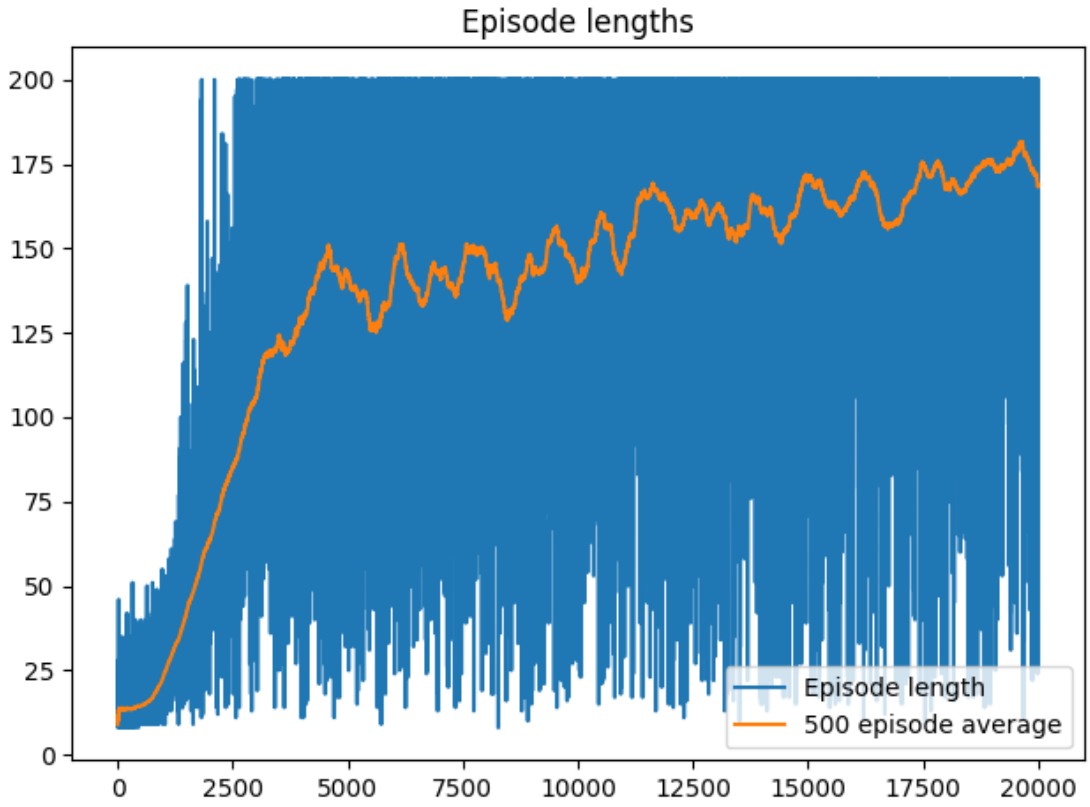

Figure 1: Task 1 Q-learning with epsilon greedy, $\epsilon = 0.2$, 20000 episodes

```
values = np.amax(q_grid, axis=4)
.
.
.
seab.heatmap(np.mean(values, axis=(1, 3)),xticklabels=True,
    yticklabels=True)

plt.xlabel("X")
plt.ylabel("theta")
plt.title('heatmap')
plt.show()
```

For the plots for task 2 please see Figures 3, 4, 5, 6

3 Question 1

1. **before the training** - Figure 3 shows that in the beginning the value function is zero for all states, because the environment is completely unexplored.

2. **after a single episode** - In Figure 4 we can see that after one episode the agent

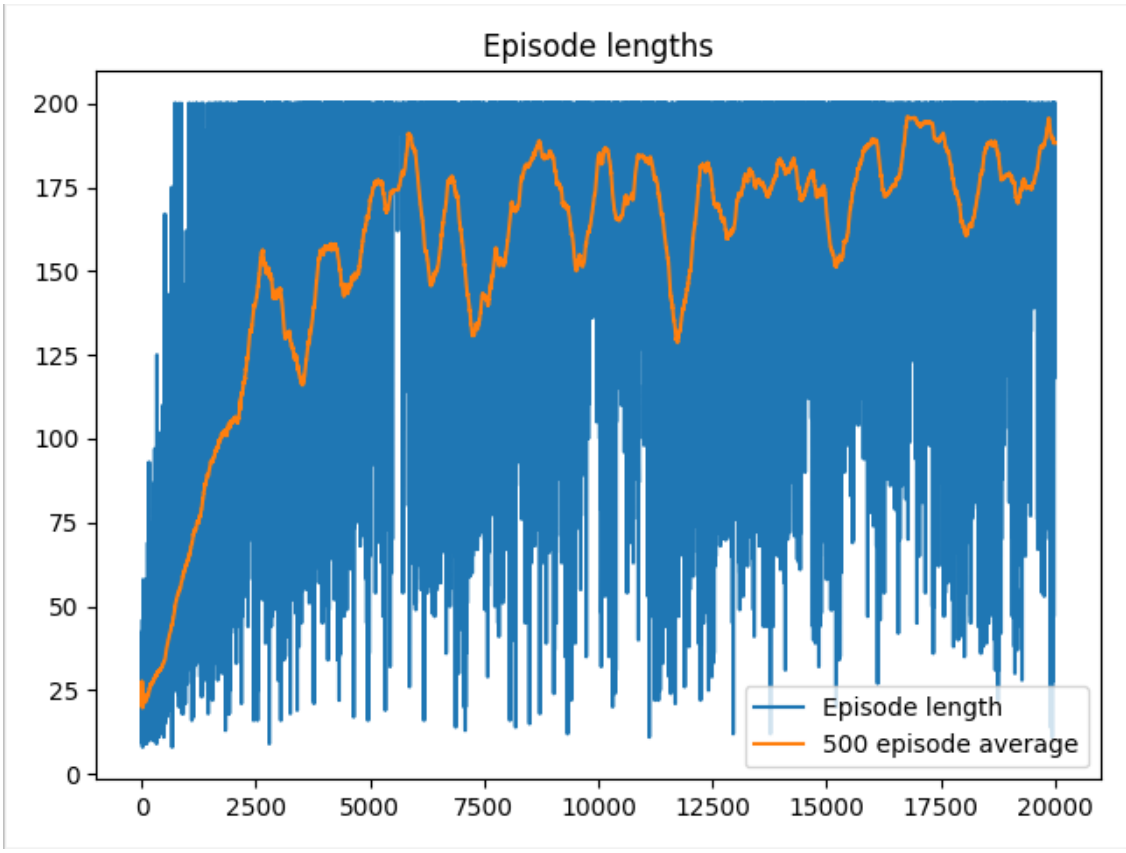

Figure 2: Task 1 Q-learning GLIE with desired epsilon, $\epsilon = 0.1$, 20000 episodes

started to explore the environment, and there are some non zero values cells.

3. **halfway through the training** - After the halfway of training we can see from Figure 5 that agent explored more states of the environment, which values are non zero, however there are black pixels (unexplored states), because the agent hasn't been in Q-table cell corresponding to those state and action.

Figure 6 shows the map of states after the 20000 episodes of training.

4 Task 3

Figures 7 and 8 shows performance of training starting from initial Q value 0 and 50.

5 Question 2.1

The model has a better performance starting with the initial Q values 50, for each state-action pair.

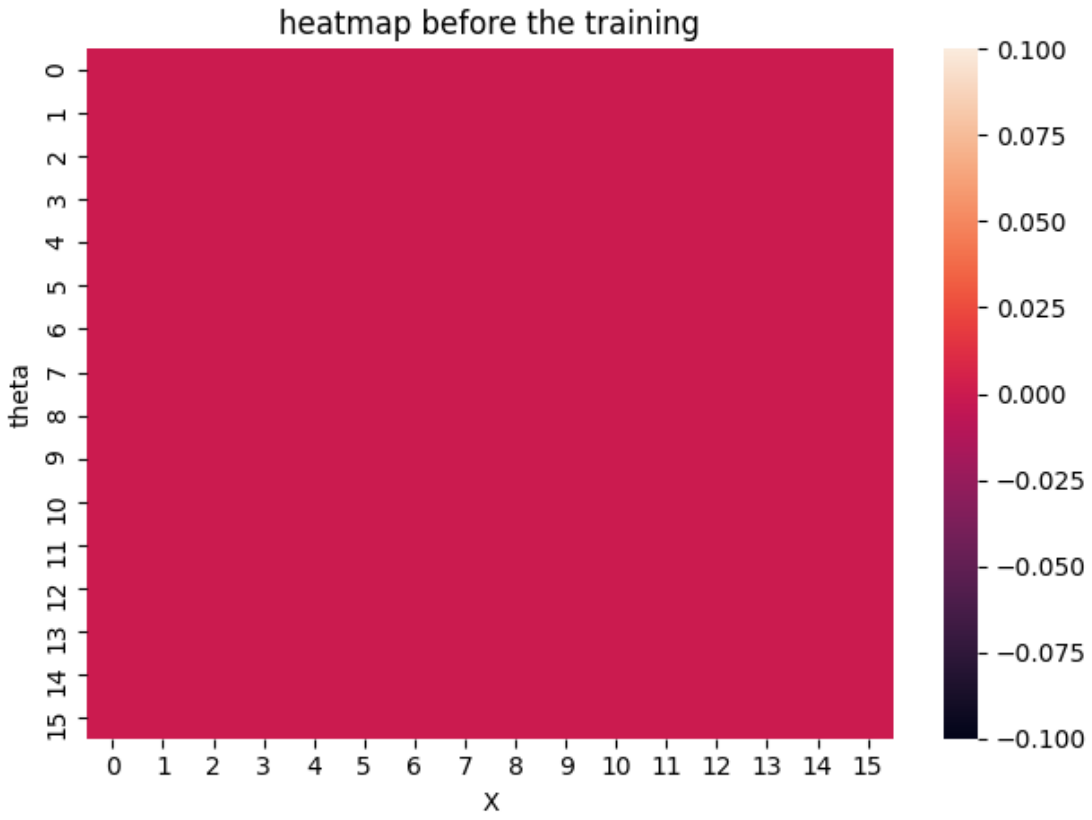

Figure 3: Task 2 a) Optimal value function before the training

6 Question 2.2

The agent performs better with higher initial Q values because each state-action pair will be chosen by agent at least once, in the beginning. For example, when the agent will take an action $action = np.argmax(q\_values[state])$, each action will have the same chance, but after some iterations, the Q value for state-action pair that did not lead to reward, will decrease, and the agent will take another action beeing in the same state. This encourages the exploration, the agent discover the best action-value pairs faster.

7 Task 4. Lunar lander

On this task the code from task 1 was modified.
In the following code, the limits of the state space are defined

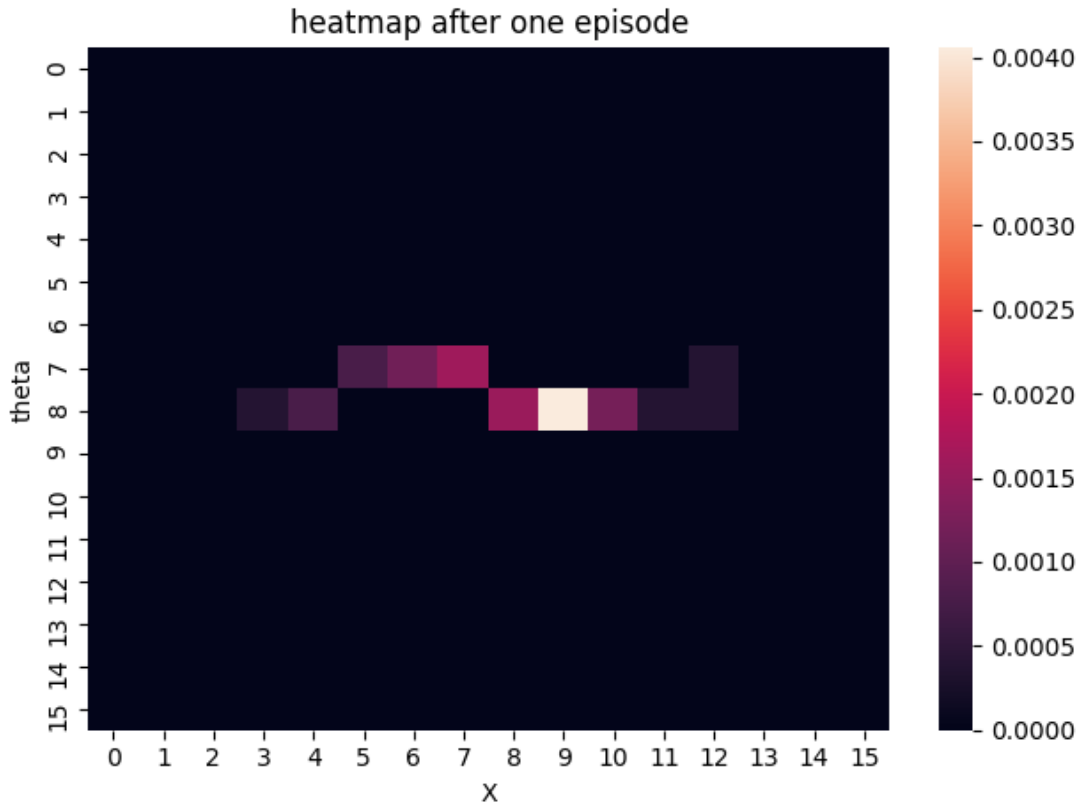

Figure 4: Task 2 b) Optimal value function after 1 episode

```
x_min, x_max = -1.2, 1.2
y_min, y_max = -.3, 1.2
xdot_min, xdot_max = -2.4, 2.4
ydot_min, ydot_max = -2., 2.
theta_min, theta_max = -6.28, 6.28
thetadot_min, thetadot_max = -8., 8.
cl_min, cl_max = 0, 1
cr_min, cr_max = 0, 1
.
x_grid = np.linspace(x_min, x_max, discr)
y_grid = np.linspace(y_min, y_max, discr)
xdot_grid = np.linspace(xdot_min, xdot_max, discr)
ydot_grid = np.linspace(ydot_min, ydot_max, discr)
theta_grid = np.linspace(theta_min, theta_max, discr)
thetadot_grid = np.linspace(thetadot_min, thetadot_max, discr)
cl_grid = np.linspace(cl_min, cl_max, 2)
cr_grid = np.linspace(cr_min, cr_max, 2)
.
q_grid = np.zeros((discr, discr, discr, discr, discr, discr, 2,
    2, num_of_actions)) + initial_q  #LunarLander
```

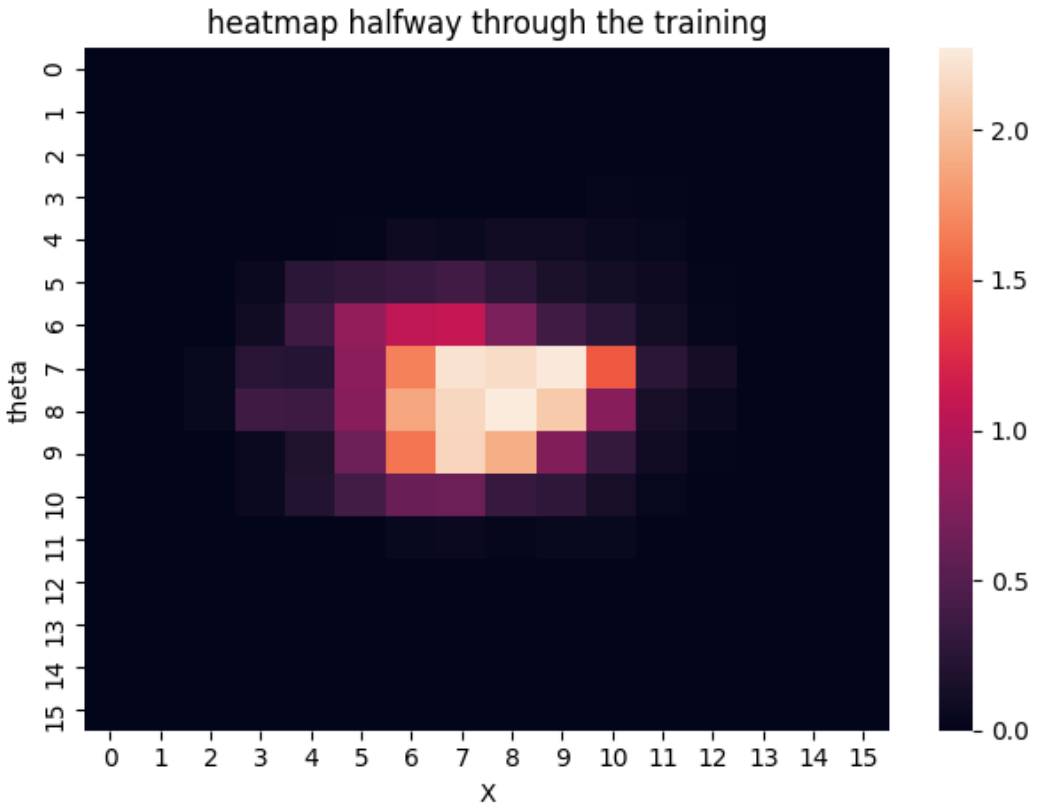

Figure 5: Task 2 c) Optimal value function halfway through the training

The function *get_cell_index(state):* was changed to:

```
#used for LunarLander
def get_cell_index(state):
    x = find_nearest(x_grid, state[0])
    y = find_nearest(y_grid, state[1])
    xdot = find_nearest(xdot_grid, state[2])
    ydot = find_nearest(ydot_grid, state[3])
    theta = find_nearest(theta_grid, state[4])
    thetadot = find_nearest(thetadot_grid, state[5])
    cl = find_nearest(cl_grid, state[6])
    cr = find_nearest(cr_grid, state[7])
    return x, y, xdot, ydot, theta, thetadot, cl, cr
```

The policy and update Q table are the same.
The performance of the agent on Luna Lander is presented in Figure 9

Figure 6: Task 2 Optimal value function after the training

8 Question 3.1

The agent was not able to learn the Luna Lander environment. This environment is more complicated that then CartPole env, and the prev setup is simple for this problem. The world discretization used in the CartPole env, is very poor for Luna Lander, the discretion of the Luna Lander cannot sufficiently cover the entire state space.
In our case, we discretize the state space into 16 bins for each feature, we may increase the number of bins for each feature for a better performance. However, it is not recommended, since we will have a lot of states, a lot of memory will be consumed, and in addition we need a lot of computational power to explore each state.
A better idea is to use function approximators for state space (Neural Networks, RBF).

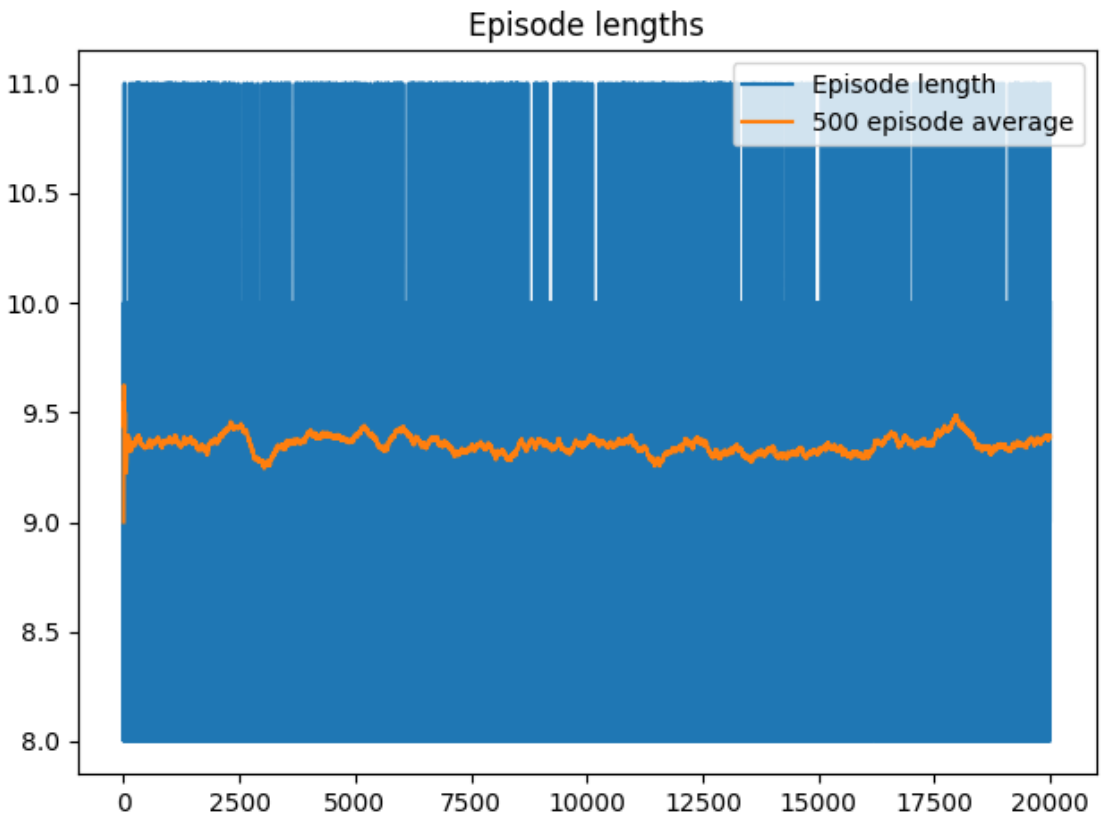

Figure 7: Task 3 Q-learning, greedy policy with Q values starting from 0

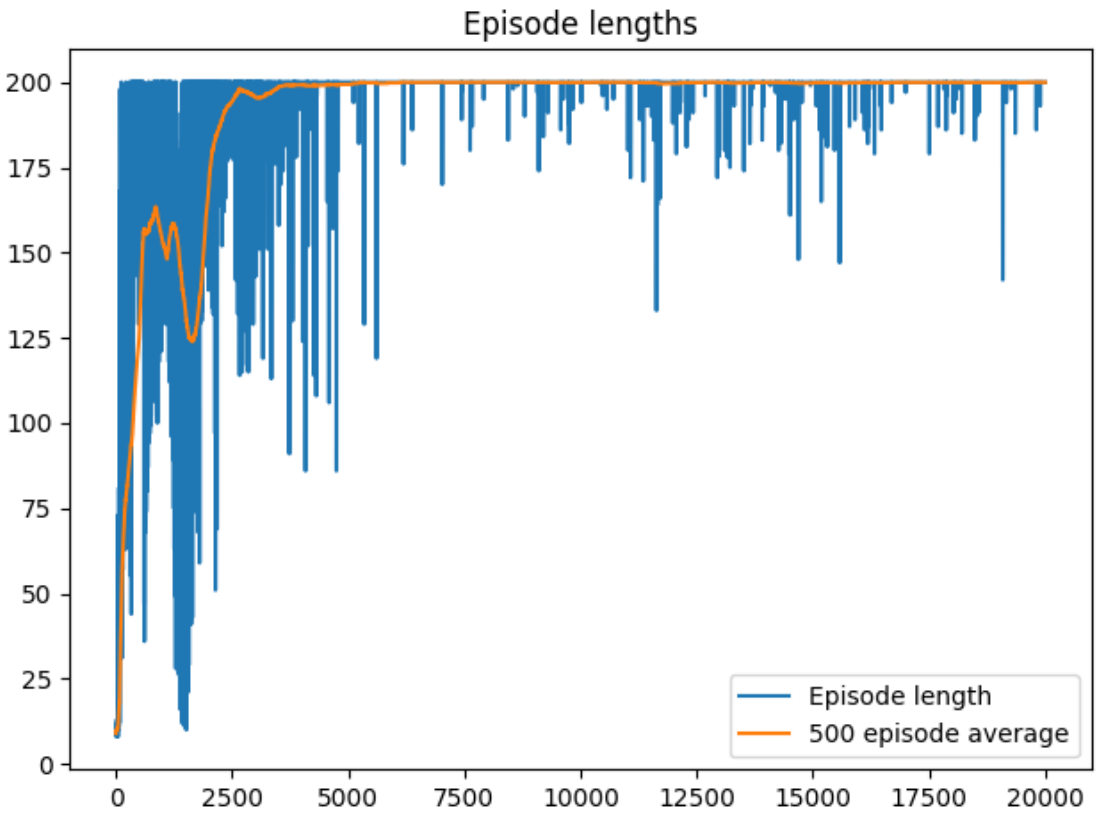

Figure 8: Task 3 Q-learning, greedy policy with Q values starting from 50

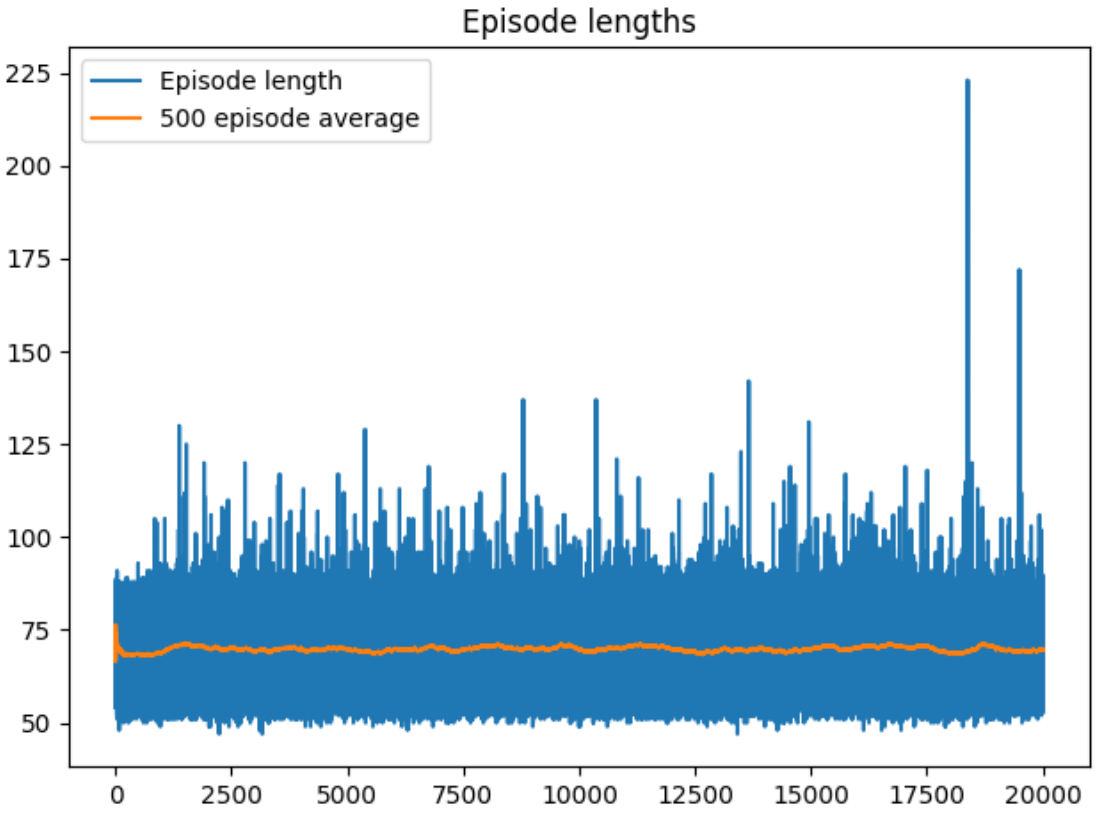

Figure 9: Task 4 Q-learning, GLIE for Luna Lander