HOMEWORK 7

Mondo Jiang
gjiang25 <9085879535>

Instructions: Use this latex file as a template to develop your homework. Please submit a single pdf to Canvas. Late submissions may not be accepted. You can choose any programming language (i.e. python, R, or MATLAB). Please check Piazza for updates about the homework.

https://github.com/MondoGao/uwm-cs760-hw7

1 Getting Started

Before you can complete the exercises, you will need to setup the code. In the zip file given with the assignment, there is all of the starter code you will need to complete it. You will need to install the requirements.txt where the typical method is through python's virtual environments. Example commands to do this on Linux/Mac are:

```
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

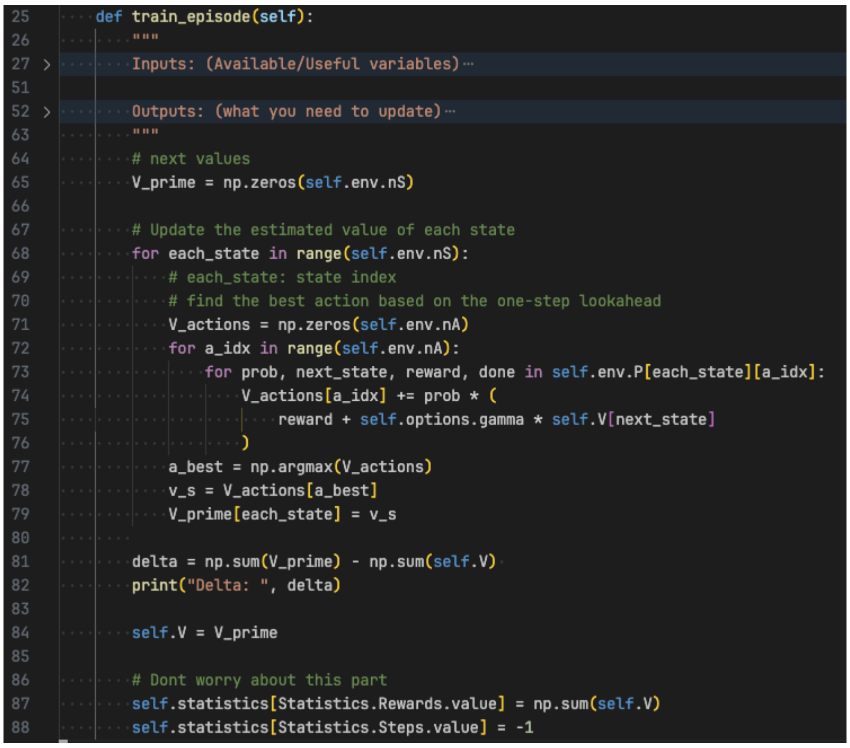
For Windows or more explanation see here: https://docs.python.org/3/tutorial/venv.html

2 Value Iteration [40 pts]

The `ValueIteration` class in `solvers/Value_Iteration.py` contains the implementation for the value iteration algorithm. Complete the `train_episode` and `create_greedy_policy` methods.

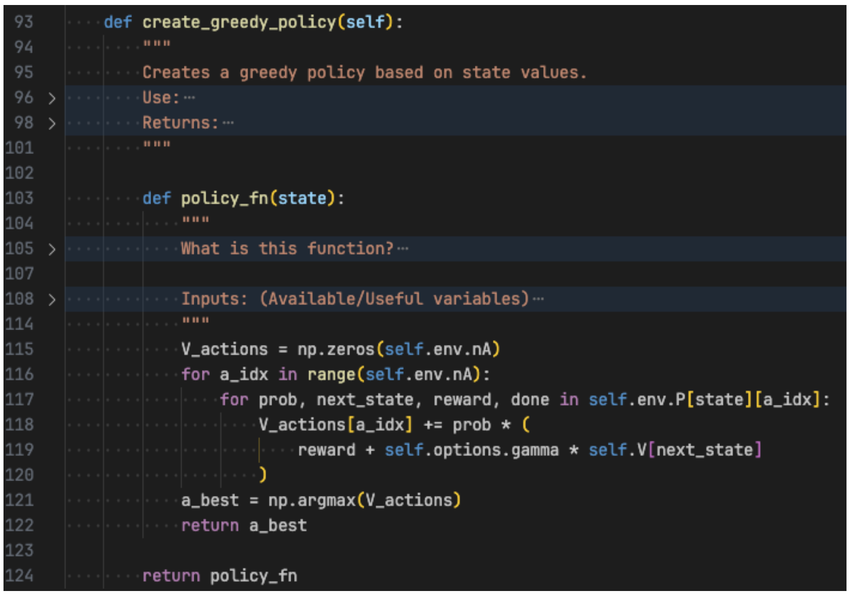
Submission [6 pts each + 10 pts for code submission]

Submit a screenshot containing your `train_episode` and `create_greedy_policy` methods (10 points).



```python
    def train_episode(self):
        """
        Inputs: (Available/Useful variables)···
        Outputs: (what you need to update)···
        """
        # next values
        V_prime = np.zeros(self.env.nS)

        # Update the estimated value of each state
        for each_state in range(self.env.nS):
            # each_state: state index
            # find the best action based on the one-step lookahead
            V_actions = np.zeros(self.env.nA)
            for a_idx in range(self.env.nA):
                for prob, next_state, reward, done in self.env.P[each_state][a_idx]:
                    V_actions[a_idx] += prob * (
                        reward + self.options.gamma * self.V[next_state]
                    )
            a_best = np.argmax(V_actions)
            v_s = V_actions[a_best]
            V_prime[each_state] = v_s

        delta = np.sum(V_prime) - np.sum(self.V)
        print("Delta: ", delta)

        self.V = V_prime

        # Dont worry about this part
        self.statistics[Statistics.Rewards.value] = np.sum(self.V)
        self.statistics[Statistics.Steps.value] = -1
```

train_episode



```python
    def create_greedy_policy(self):
        """
        Creates a greedy policy based on state values.
        Use:···
        Returns:···
        """

        def policy_fn(state):
            """
            What is this function?···

            Inputs: (Available/Useful variables)···
            """
            V_actions = np.zeros(self.env.nA)
            for a_idx in range(self.env.nA):
                for prob, next_state, reward, done in self.env.P[state][a_idx]:
                    V_actions[a_idx] += prob * (
                        reward + self.options.gamma * self.V[next_state]
                    )
            a_best = np.argmax(V_actions)
            return a_best

        return policy_fn
```

create_greedy_policy

For these 5 commands. Report the episode it converges at and the reward it achieves. See examples for what we expect. An example is:

```
python run.py -s vi -d Gridworld -e 200 -g 0.2
```

Converges to a reward of ____ in ____ episodes.
Note: For FrozenLake the rewards go to many decimal places. Report convergence to the nearest 0.0001.

Submission Commands:

1. python run.py -s vi -d Gridworld -e 200 -g 0.05

   Converges to a reward of `-14.51` in `3` episodes.

2. python run.py -s vi -d Gridworld -e 200 -g 0.2

   Converges to a reward of `-16.16` in `3` episodes.

3. python run.py -s vi -d FrozenLake-v0 -e 500 -g 0.5

   Converges to a reward of `0.6374` in `49` episodes.

4. python run.py -s vi -d FrozenLake-v0 -e 500 -g 0.9

   Converges to a reward of `2.1761` in `261` episodes.

   Note: if we consider to stop our algorithm after change of reward equals to 0, we'll converge at `252` episodes.

5. python run.py -s vi -d FrozenLake-v0 -e 500 -g 0.75

   Converges to a reward of `1.1316` in `106` episodes.

Examples

For each of these commands. The expected reward is given for a correct solution. If your solution gives the same reward it doesn't guarantee correctness on the test cases that you report results on – you're encouraged to develop your own test cases to supplement the provided ones.

```
python run.py -s vi -d Gridworld -e 100 -g 0.9
```

Converges in 3 episodes with reward of -26.24.

```
python run.py -s vi -d Gridworld -e 100 -g 0.4
```

Converges in 3 episodes with reward of -18.64.

```
python run.py -s vi -d FrozenLake-v0 -e 100 -g 0.9
```

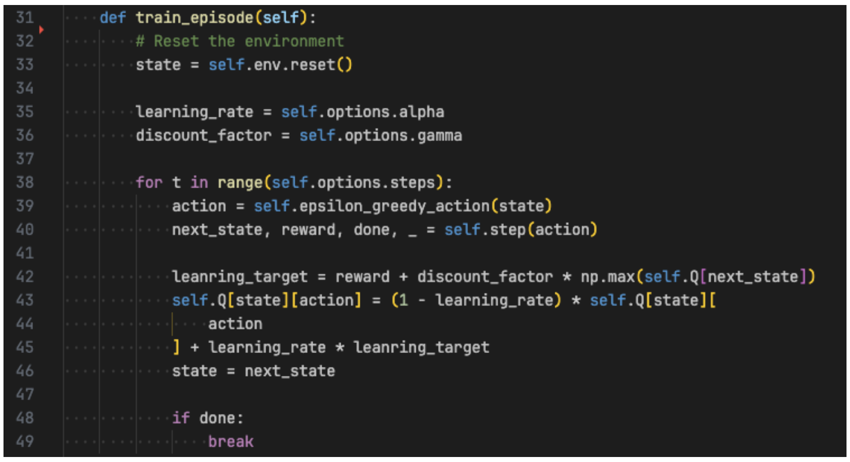
Achieves a reward of 2.176 after 53 episodes.

3 Q-learning [40 pts]

The `QLearning` class in `solvers\Q_Learning.py` contains the implementation for the Q-learning algorithm. Complete the `train_episode`, `create_greedy_policy`, and `make_epsilon_greedy_policy` methods.

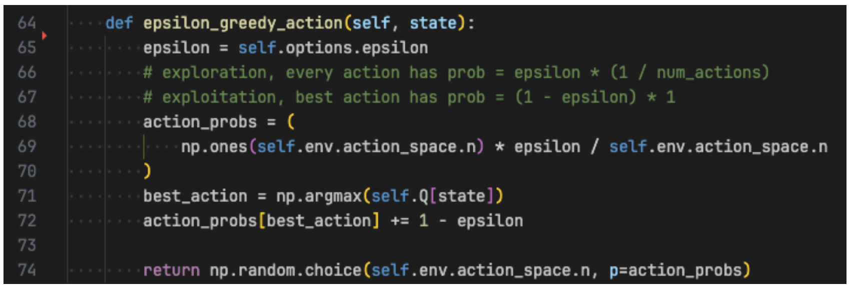
Submission [10 pts each + 10 pts for code submission]

Submit a screenshot containing your `train_episode`, `create_greedy_policy` and `make_epsilon_greedy_policy` methods (10 points).



```python
    def train_episode(self):
        # Reset the environment
        state = self.env.reset()

        learning_rate = self.options.alpha
        discount_factor = self.options.gamma

        for t in range(self.options.steps):
            action = self.epsilon_greedy_action(state)
            next_state, reward, done, _ = self.step(action)

            leanring_target = reward + discount_factor * np.max(self.Q[next_state])
            self.Q[state][action] = (1 - learning_rate) * self.Q[state][
                action
            ] + learning_rate * leanring_target
            state = next_state

            if done:
                break
```

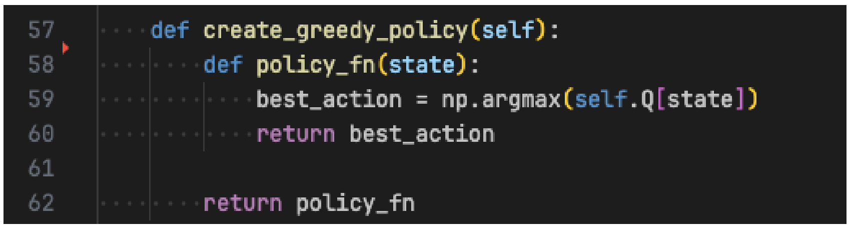
train_episode



```python
    def epsilon_greedy_action(self, state):
        epsilon = self.options.epsilon
        # exploration, every action has prob = epsilon * (1 / num_actions)
        # exploitation, best action has prob = (1 - epsilon) * 1
        action_probs = (
            np.ones(self.env.action_space.n) * epsilon / self.env.action_space.n
        )
        best_action = np.argmax(self.Q[state])
        action_probs[best_action] += 1 - epsilon

        return np.random.choice(self.env.action_space.n, p=action_probs)
```

epsilon_greedy_action

* I change the interface of this function to return action directly instead of action probs.

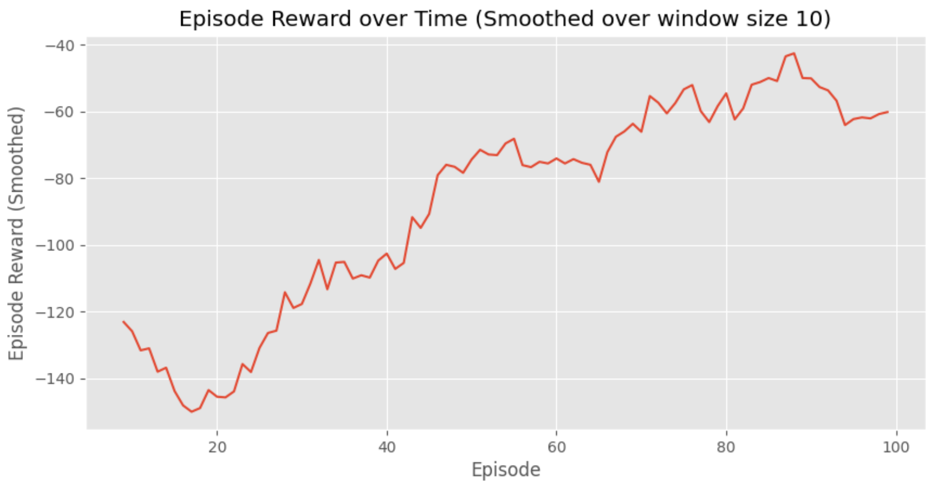


```python
    def create_greedy_policy(self):
        def policy_fn(state):
            best_action = np.argmax(self.Q[state])
            return best_action

        return policy_fn
```

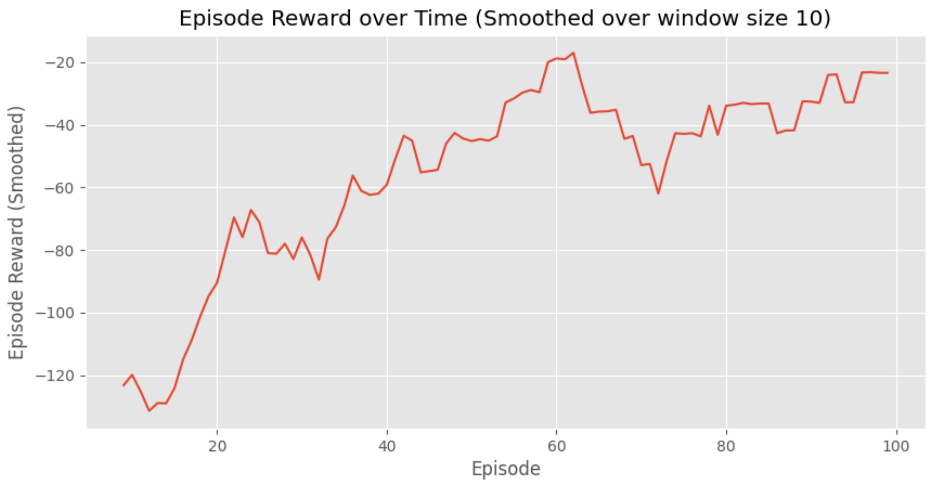
create_greedy_policy

Report the reward for these 3 commands with your implementation (10 points each) by submitting the "Episode Reward over Time" plot for each command:

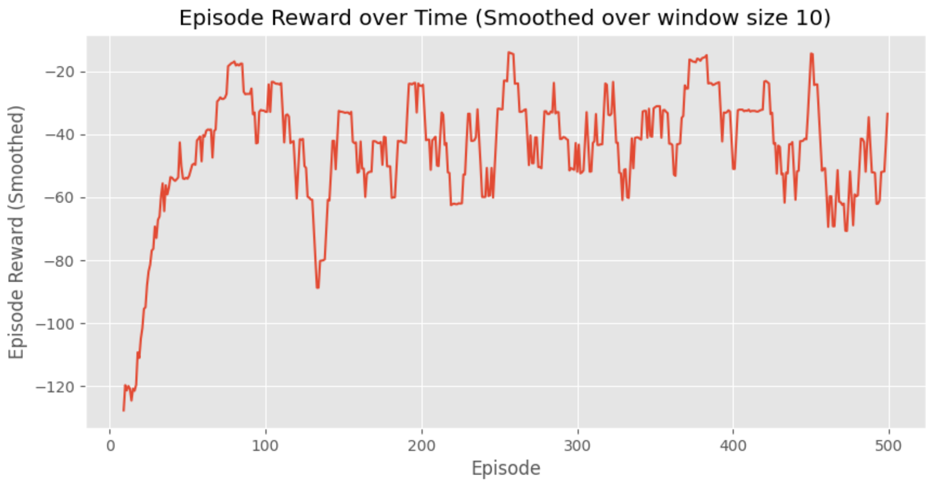
1. python run.py -s ql -d CliffWalking -e 100 -a 0.2 -g 0.9 -p 0.1



2. python run.py -s ql -d CliffWalking -e 100 -a 0.8 -g 0.5 -p 0.1



3. python run.py -s ql -d CliffWalking -e 500 -a 0.6 -g 0.8 -p 0.1



For reference, command 1 should end with a reward around -60, command 2 should end with a reward around -25 and command 3 should end with a reward around -40.

Example

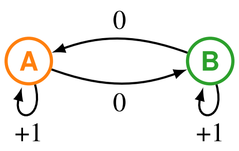
Again for this command, the expected reward is given for a correct solution. If your solution gives the same reward it doesn't guarantee correctness on the test cases.

```
python run.py -s ql -d CliffWalking -e 500 -a 0.5 -g 1.0 -p 0.1
```

Achieves a best performing policy with -13 reward.

4 Q-learning [20 pts]

For this question you can either reimplement your Q-learning code or use your previous implementation. You will be using a custom made MDP for analysis. Consider the following Markov Decision Process. It has two states s. It has two actions a: move and stay. The state transition is deterministic: "move" moves to the other state, while "stay' stays at the current state. The reward r is 0 for move, 1 for stay. There is a discounting factor $\gamma = 0.8$.



The reinforcement learning agent performs Q-learning. Recall the Q table has entries $Q(s,a)$. The Q table is initialized with all zeros. The agent starts in state $s_1 = A$. In any state s_t, the agent chooses the action a_t according to a behavior policy $a_t = \pi_B(s_t)$. Upon experiencing the next state and reward s_{t+1}, r_t the update is:

$$Q(s_t, a_t) \Leftarrow (1-\alpha)Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a'} Q(s_{t+1}, a') \right).$$

Let the step size parameter $\alpha = 0.5$.

1. (5 pts) Run Q-learning for 200 steps with a deterministic greedy behavior policy: at each state s_t use the best action $a_t \in \arg\max_a Q(s_t, a)$ indicated by the current action-value table. If there is a tie, prefer move. Show the action-value table at the end.

State/Action	Move	Stay
A	0	0
B	0	0

Q table after 200 steps within 1 episode

Because we prefer move when there is a tie, the algorithm is always jumping between A and B, and we'll always get a 0 learning target, there is no surprise the action-value table are all zeros.

2. (5 pts) Reset and repeat the above, but with an ϵ-greedy behavior policy: at each state s_t, with probability $1-\epsilon$ choose what the current Q table says is the best action: $\arg\max_a Q(s_t, a)$; Break ties arbitrarily. Otherwise, (with probability ϵ) uniformly chooses between move and stay (move or stay both with 1/2 probability). Use $\epsilon = 0.5$.

State/Action	Move	Stay
A	3.9980	4.9983
B	3.9980	4.9983

Q table after 200 steps within 1 episode

3. (5 pts) Without doing simulation, use Bellman equation to derive the true action-value table induced by the MDP. That is, calculate the true optimal action-values by hand.

We have:

$$S = A, B$$

$$A(s) = \{move, stay\}$$

$$P(s'|s,a) = \begin{cases} 1 & \text{if } s' = s \text{ and } a = stay \\ 1 & \text{if } s' \neq s \text{ and } a = move \\ 0 & \text{otherwise} \end{cases}$$

$$r(s,a) = \begin{cases} 1 & \text{if } a = stay \\ 0 & \text{if } a = move \end{cases}$$

Applying Bellman equation, we have:

$$Q(A, move) = \sum_{s' \in S} P(s'|A, move)(r(s, move) + \gamma V(s')) = 0.8V(B) = 0.8 \max(Q(B, move), Q(B, stay))$$

Since reward for stay is 1 and reward for move is 0, we can easily know to maximize the total rewards we'll always stay at s_0, then we have $Q(a, stay) \geq Q(a, move)$, so:

$$Q(A, move) = 0.8Q(B, stay)$$

Likewise,

$$Q(A, stay) = \sum_{s' \in S} P(s'|A, stay)(r(s, stay) + \gamma V(s')) = 1 + 0.8Q(A, stay)$$

$$Q(B, stay) = \sum_{s' \in S} P(s'|B, stay)(r(s, stay) + \gamma V(s')) = 1 + 0.8Q(B, stay)$$

$$Q(A, stay), Q(B, stay) = 5$$

Substitue $Q(A, stay)$ and $Q(B, stay)$ into $Q(A, move)$, $Q(B, move)$, we have:

$$Q(B, move) = \sum_{s' \in S} P(s'|B, move)(r(s, move) + \gamma V(s')) = 0.8Q(A, stay)$$

$$Q(A, move), Q(B, move) = 4$$

State/Action	Move	Stay
A	4	5
B	4	5

Q table

4. (5 pts) To the extent that you obtain different solutions for each question, explain why the action-values differ.

For first method, because we prefer to move when there is a tie, the algorithm is always jumping between A and B, and gain no reward during the update, so the action-values are all zeros.

For second method, we have a ϵ-greedy behavior policy, so we'll have a chance to explore the environment, and we'll get a reward of 1 when we stay at s_0, so we can get a action-values near real distribution.

Because we actually know the transition probabilities and rewards, we can easily get the optimial policy, and we can see our result from q-learning is very close to the optimial solution.

5 A2C (Extra credit)

5.1 Implementation

You will implement a function for the A2C algorithm in solvers/A2C.py. Skeleton code for the algorithm is already provided in the relevant python files. Specifically, you will need to complete `train` for A2C. To test your implementation, run:

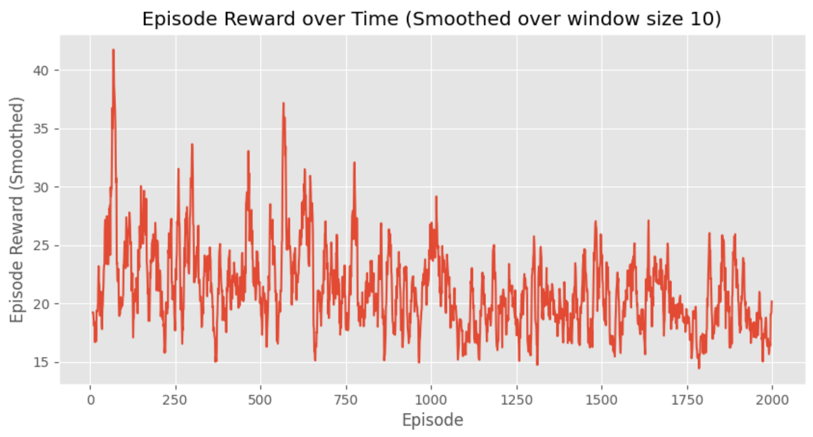
```
python run.py -s a2c -t 1000 -d CartPole-v1 -G 200

-e 2000 -a\ 0.001 -g 0.95 -l [32]
```

This command will train a neural network policy with A2C on the CartPole domain for 2000 episodes. The policy has a single hidden layer with 32 hidden units in that layer.

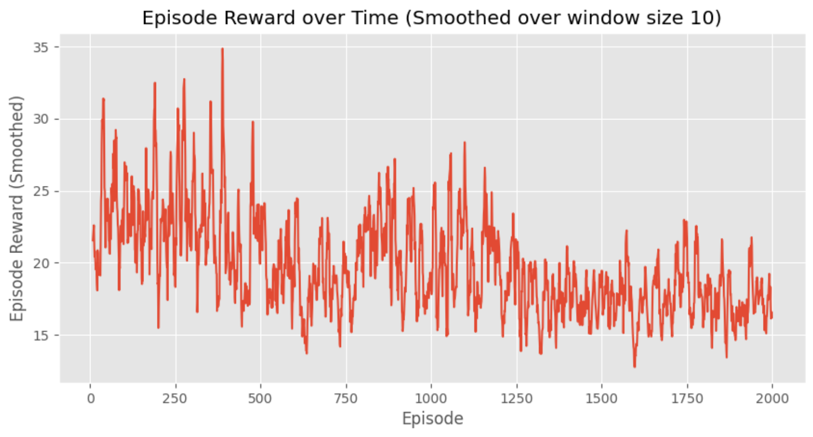
Submission

For submission, plot the final reward/episode for 5 different values of either alpha or gamma. Then include a short (<`5 sentence`) analysis on the impact that alpha/gamma had for the reward in this domain.

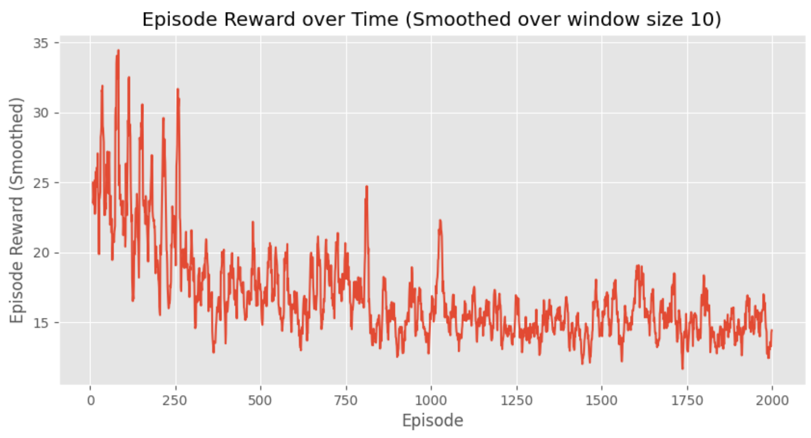
Increacing gamma ($\gamma \in (0.5, 0.8, 0.9, 0.99)$**,** $\alpha = 0.001$)



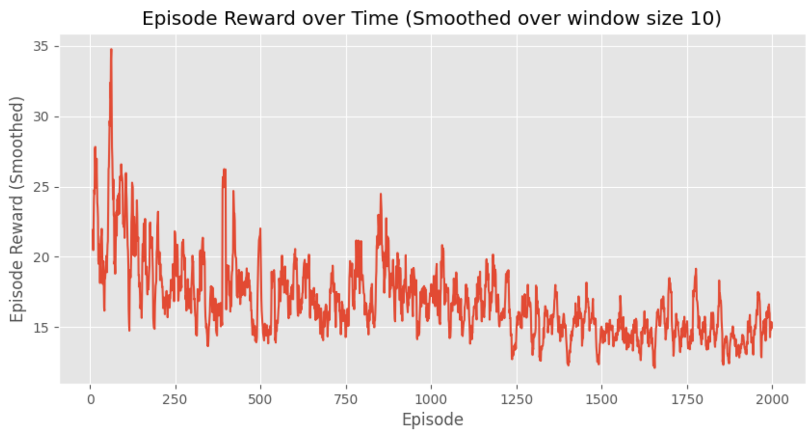
gamma=0.5, alpha=0.001



gamma=0.8, alpha=0.001



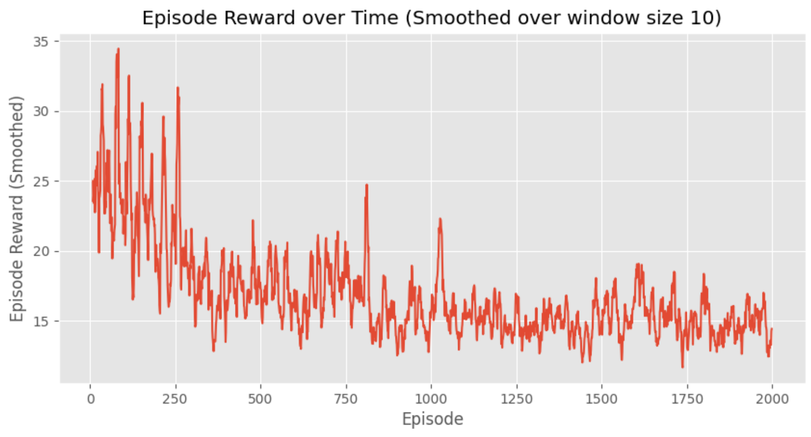
gamma=0.9, alpha=0.001



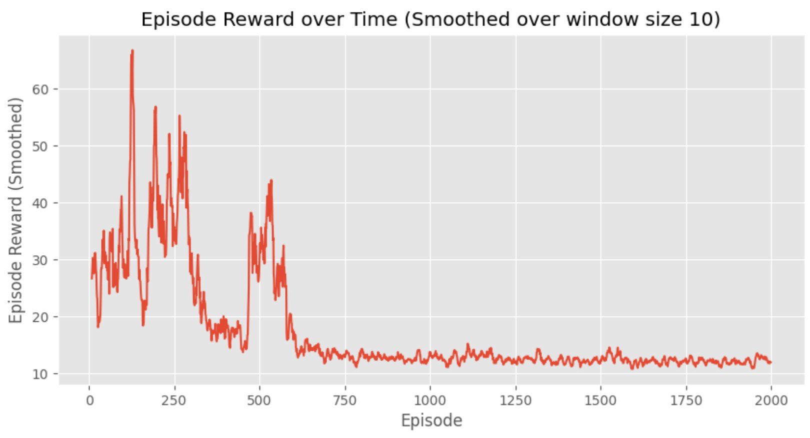
gamma=0.99, alpha=0.001

With gamma increasing, the agent will consider more about future rewards.

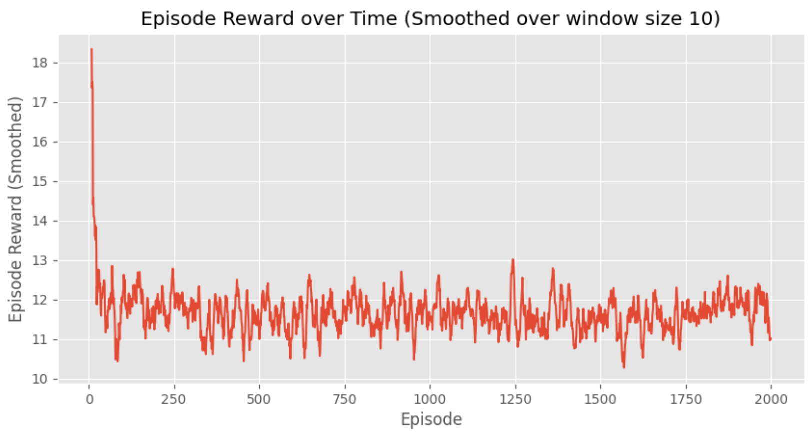
Increacing alpha ($\gamma = 0.9$, $\alpha \in (0.001, 0.01, 0.1)$)



gamma=0.9, alpha=0.001



gamma=0.9, alpha=0.01



gamma=0.9, alpha=0.1

With alpha increasing, the agent will learn faster, but it will sometimes became unstable. Like when alpha=0.01, the reward suddenly increace around episode 500, but this doesn't happen in every training.