HOMEWORK 7

Harry Zhang
hzhang699

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.

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).

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
2. python run.py -s vi -d Gridworld -e 200 -g 0.2
3. python run.py -s vi -d FrozenLake-v0 -e 500 -g 0.5
4. python run.py -s vi -d FrozenLake-v0 -e 500 -g 0.9
5. python run.py -s vi -d FrozenLake-v0 -e 500 -g 0.75

1. Converges to a reward of -14.51 in 3 episodes.
2. Converges to a reward of -16.16 in 3 episodes.

3. Converges to a reward of 0.6374 in 10 episodes.
4. Converges to a reward of 2.1761 in 57 episodes.
5. Converges to a reward of 1.1316 in 21 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]

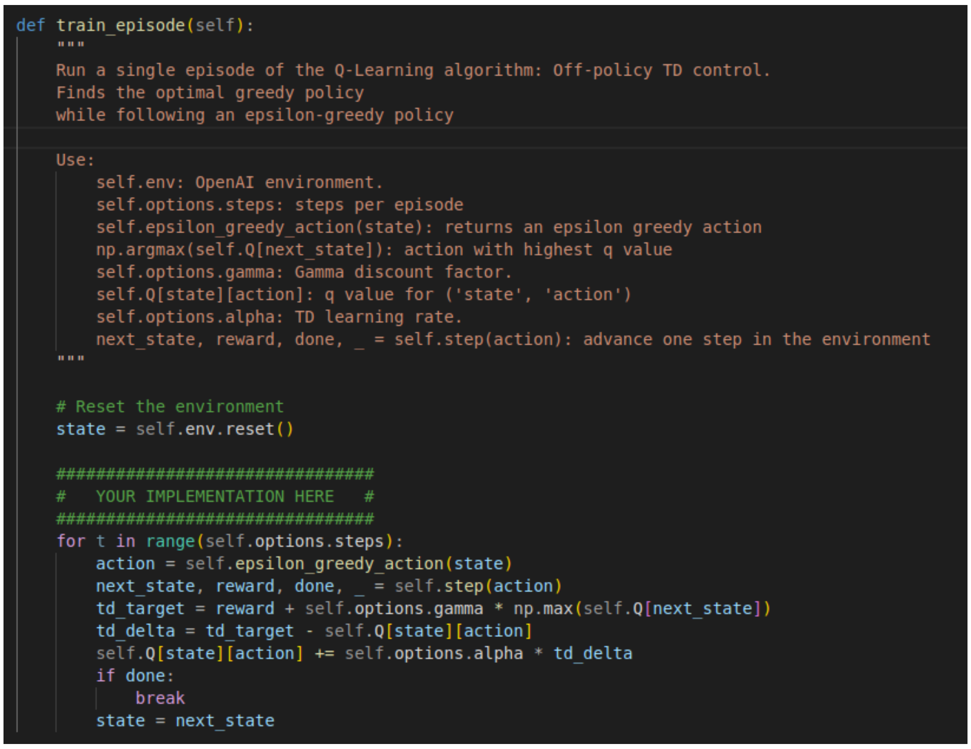
Submit a screenshot containing your `train_episode`, `create_greedy_policy` and `make_epsilon_greedy_policy` methods (10 points).
Report the reward for these 3 commands with your implementation (10 points each) by submitting the "Episode Reward over Time" plot for each command:

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.

1. Ended around -60 reward.
2. Ended around -30 reward.
3. Ended around -40 reward.

The following figures are the code screenshots as in Fig. 1, 2.



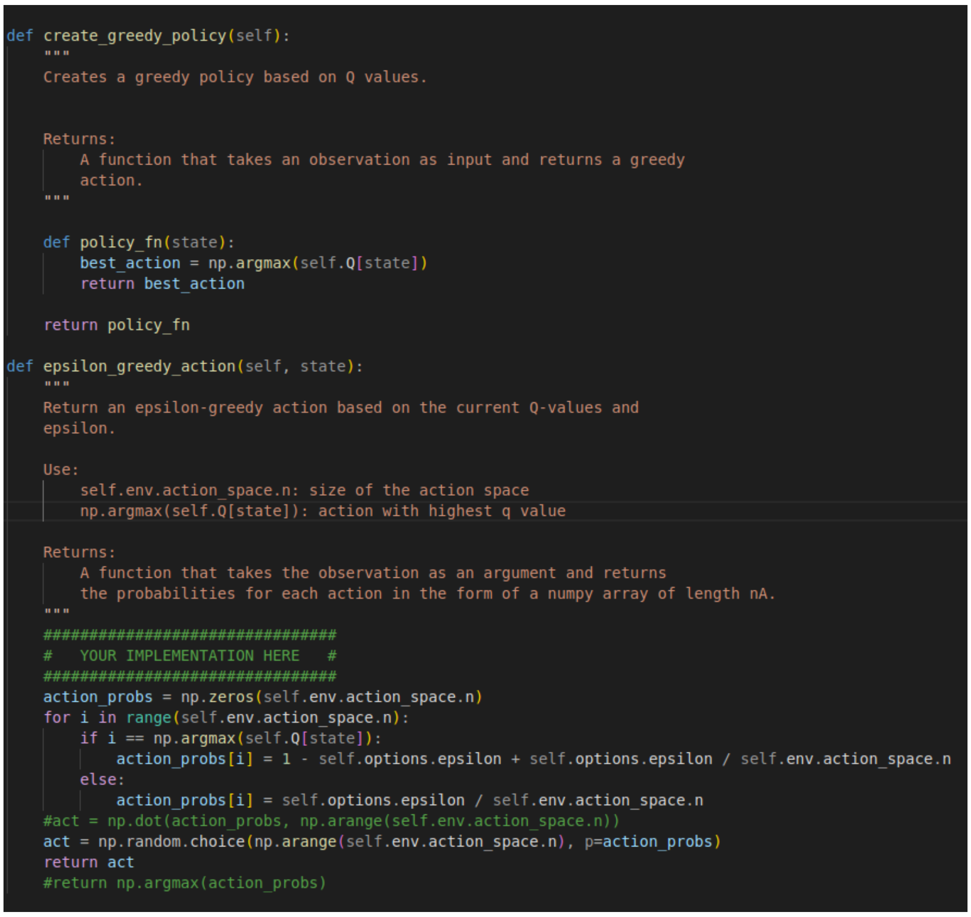
```python
def train_episode(self):
    """
    Run a single episode of the Q-Learning algorithm: Off-policy TD control.
    Finds the optimal greedy policy
    while following an epsilon-greedy policy

    Use:
        self.env: OpenAI environment.
        self.options.steps: steps per episode
        self.epsilon_greedy_action(state): returns an epsilon greedy action
        np.argmax(self.Q[next_state]): action with highest q value
        self.options.gamma: Gamma discount factor.
        self.Q[state][action]: q value for ('state', 'action')
        self.options.alpha: TD learning rate.
        next_state, reward, done, _ = self.step(action): advance one step in the environment
    """

    # Reset the environment
    state = self.env.reset()

    ################################
    #   YOUR IMPLEMENTATION HERE   #
    ################################
    for t in range(self.options.steps):
        action = self.epsilon_greedy_action(state)
        next_state, reward, done, _ = self.step(action)
        td_target = reward + self.options.gamma * np.max(self.Q[next_state])
        td_delta = td_target - self.Q[state][action]
        self.Q[state][action] += self.options.alpha * td_delta
        if done:
            break
        state = next_state
```

Figure 1: Code for train_episode



```python
def create_greedy_policy(self):
    """
    Creates a greedy policy based on Q values.


    Returns:
        A function that takes an observation as input and returns a greedy
        action.
    """

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

    return policy_fn

def epsilon_greedy_action(self, state):
    """
    Return an epsilon-greedy action based on the current Q-values and
    epsilon.

    Use:
        self.env.action_space.n: size of the action space
        np.argmax(self.Q[state]): action with highest q value

    Returns:
        A function that takes the observation as an argument and returns
        the probabilities for each action in the form of a numpy array of length nA.
    """
    ################################
    #   YOUR IMPLEMENTATION HERE   #
    ################################
    action_probs = np.zeros(self.env.action_space.n)
    for i in range(self.env.action_space.n):
        if i == np.argmax(self.Q[state]):
            action_probs[i] = 1 - self.options.epsilon + self.options.epsilon / self.env.action_space.n
        else:
            action_probs[i] = self.options.epsilon / self.env.action_space.n
    #act = np.dot(action_probs, np.arange(self.env.action_space.n))
    act = np.random.choice(np.arange(self.env.action_space.n), p=action_probs)
    return act
    #return np.argmax(action_probs)
```

Figure 2: Code for create_greedy_policy and make_epsilon_greedy_policy

The following are the plots for reward vs episode as in Fig. 3, 4, 5.

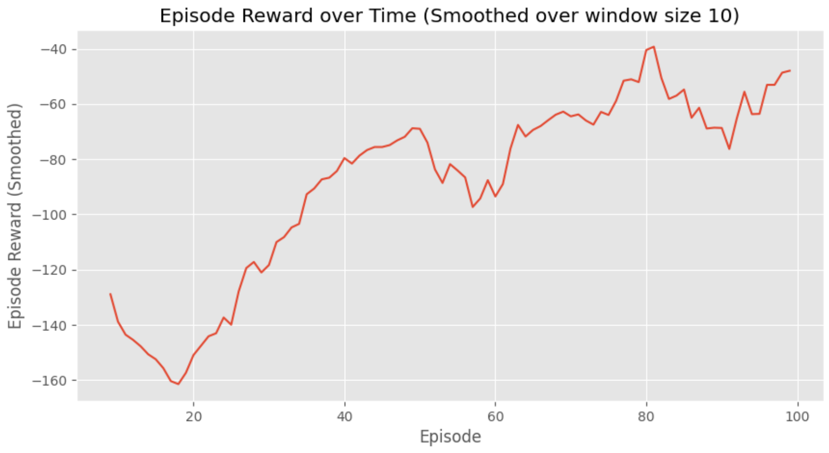


Figure 3: Result 1

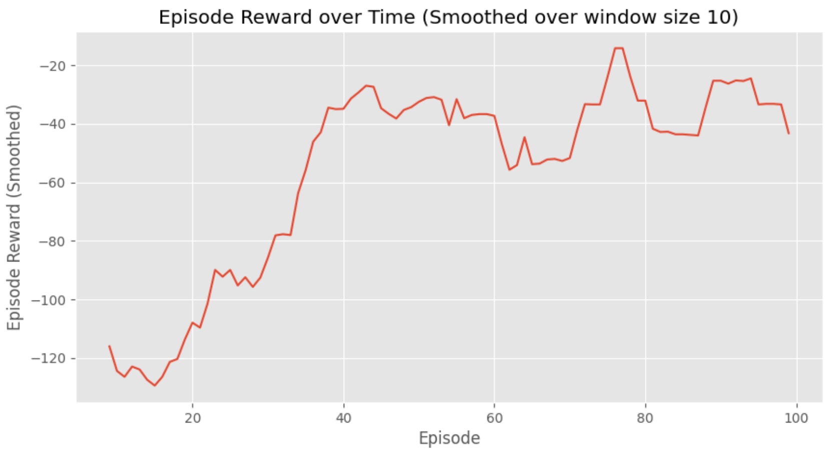


Figure 4: Result 2

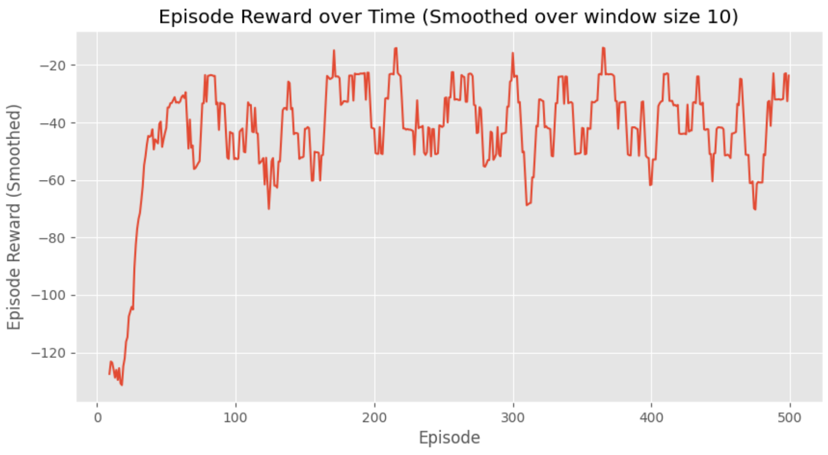
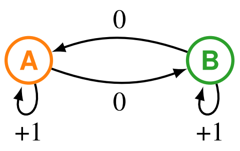


Figure 5: Result 3

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. The action-value table at the end is shown in Table 1.

State	Move	Stay
A	0.0	0.0
B	0.0	0.0

Table 1: Action-value table

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 $\epsilon = 0.5$. The action-value table at the end is shown in Table 2.

State	Move	Stay
A	2.5	4.5
B	1.8	4.0

Table 2: Action-value table

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.

$$\begin{aligned}Q(s,a) &= \sum_{s'}\sum_{r} p(s',r|s,a)[r + \gamma v_\pi(s')]\\ &= \sum_{s'}\sum_{r} p(s',r|s,a)[r + \gamma \max Q(s,:)]\\ Q(A, move) &= 1.0(0 + 0.8) = 0.8\\ Q(A, stay) &= 1.0(1 + 0.8) = 1.8\\ Q(B, move) &= 1.0(0 + 0.8) = 0.8\\ Q(B, stay) &= 1.0(1 + 0.8) = 1.8\end{aligned}$$

4. (5 pts) To the extent that you obtain different solutions for each question, explain why the action-values differ.
The action-values differ because the (1) use special split tie rule and (2) introduce randomness in the policy.

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.

The results from changing α parameters are shown in Fig. 6, 7, 8, 9, 10. By tuning the values of α, we can see that reward won't converge into good values if the learning rate, α, is too large or too small. The best value of α is 0.00075 as shown in Fig. 10.

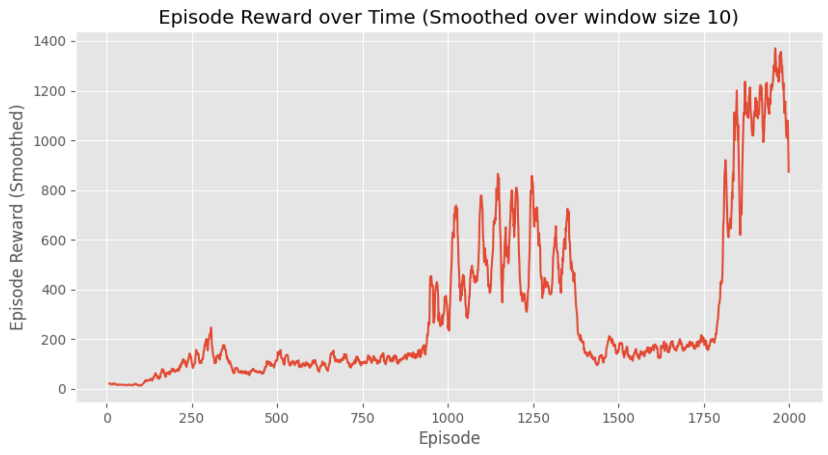


Figure 6: Reward vs Episode for $\alpha = 0.001$ $\gamma = 0.95$

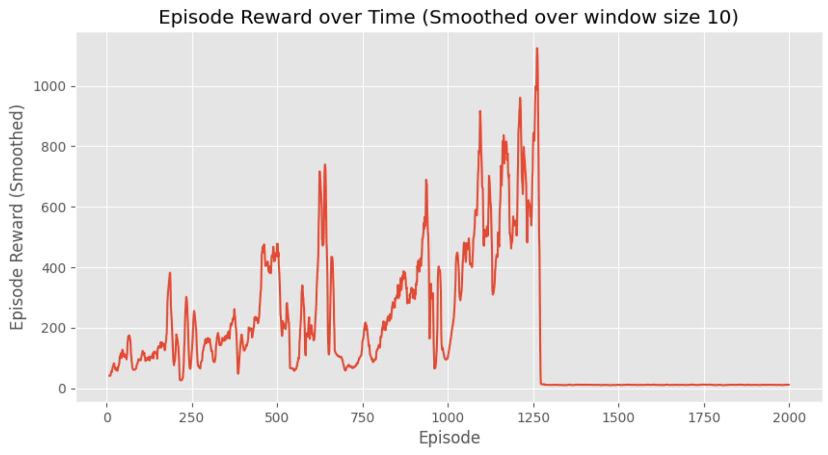


Figure 7: Reward vs Episode for $\alpha = 0.01$ $\gamma = 0.95$

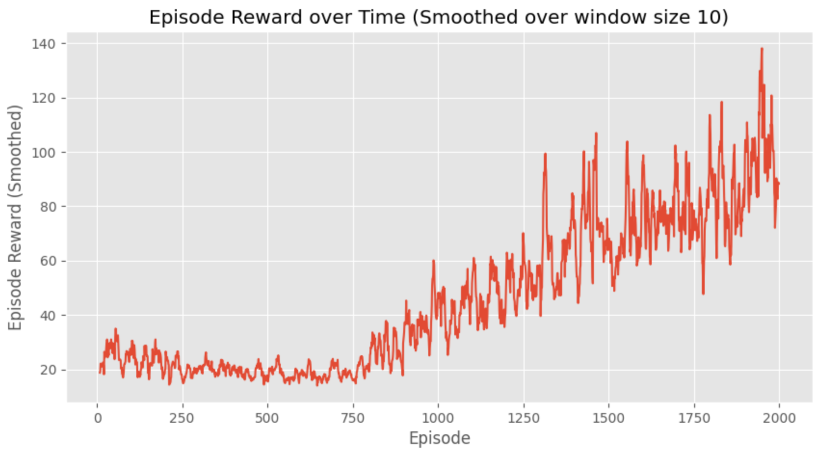


Figure 8: Reward vs Episode for $\alpha = 0.0001$ $\gamma = 0.95$

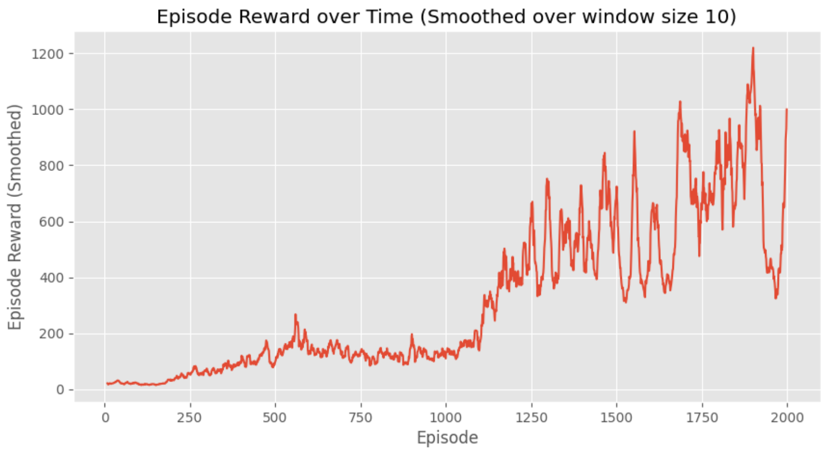


Figure 9: Reward vs Episode for $\alpha = 0.0005$ $\gamma = 0.95$

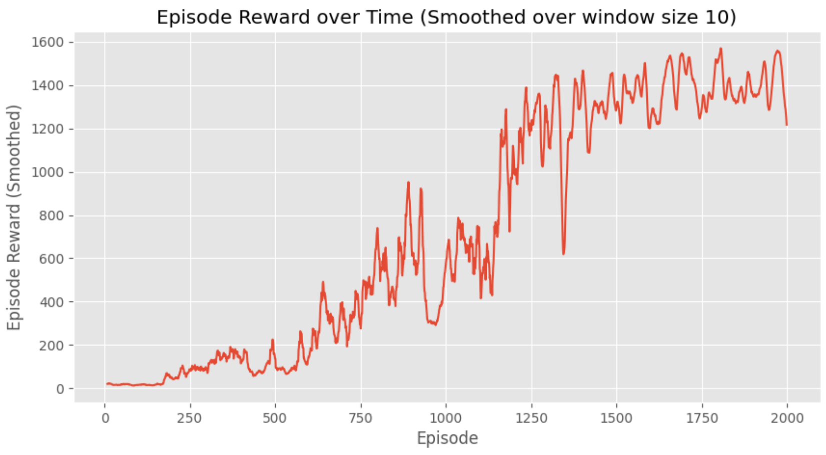


Figure 10: Reward vs Episode for $\alpha = 0.00075$ $\gamma = 0.95$