HOMEWORK 7

>>Sriram Ashokkumar<<
>>908 216 3750<<

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). Latex code to include image

```python
def create_greedy_policy(self):
    """
    Creates a greedy policy based on state values.
    Use:
        self.env.nA: Number of actions in the environment.
    Returns:
        A function that takes an observation as input and returns a Greedy
            action
    """

    def policy_fn(state):
        """
        What is this function?
            This function is the part that decides what action to take

        Inputs: (Available/Useful variables)
            self.V[state]
                the estimated long-term value of getting to a state

            self.env.nA:
                number of actions in the environment
        """

        ################################
        #   YOUR IMPLEMENTATION HERE   #
        ################################
        best_action = np.argmax([sum(prob * (reward + self.options.gamma * self.V[next_state] * (not done))
                                for prob, next_state, reward, done in self.env.P[state][action])
                                for action in range(self.env.nA)])

        return best_action

    return policy_fn
```

Figure 1: create_greedy_policy

```python
            Outputs: (what you need to update)
                self.V:
                    This is a numpy array, but you can think of it as a dictionary
                    `self.V[state]` should return a floating point value that
                    represents the value of a state. This value should become
                    more accurate with each episode.

                    How should this be calculated?
                        look at the value iteration algorithm
                        Ref: Sutton book eq. 4.10.
                    Once those values have been updated, thats it for this function/class
        """

        # you can add variables here if it is helpful

        # Update the estimated value of each state
        for each_state in range(self.env.nS):

            ###################################################
            #            Compute self.V here                  #
            # Do a one-step lookahead to find the best action #
            #           YOUR IMPLEMENTATION HERE              #
            ###################################################
            # Compute self.V here using a one-step lookahead to find the best action
            best_action_value = float("-inf")

            for action in range(self.env.nA):
                action_value = 0.0

                for probability, next_state, reward, done in self.env.P[each_state][action]:
                    action_value += probability * (reward + self.options.gamma * self.V[next_state] * (not done))

                best_action_value = max(best_action_value, action_value)

            self.V[each_state] = best_action_value

        self.statistics[Statistics.Rewards.value] = np.sum(self.V)
        self.statistics[Statistics.Steps.value] = -1
```

Figure 2: train_episode

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 10 episodes.
4. python run.py -s vi -d FrozenLake-v0 -e 500 -g 0.9 Converges to a reward of 2.1761 in 57 episodes.
5. python run.py -s vi -d FrozenLake-v0 -e 500 -g 0.75 Converges to a reward of 1.1316 in 21 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).
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.

```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)
            action = np.random.choice(np.arange(len(action)), p=action)
            next_state, reward, done, _ = self.step(action)

            td_target = (reward + self.options.gamma * np.max(self.Q[next_state]) - self.Q[state][action])

            self.Q[state][action] += self.options.alpha * td_target

            if done:
                break

            state = next_state
```

```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
```

```python
    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   #
        ################################
        prob = np.zeros(self.env.action_space.n)

        for curAction in range(self.env.action_space.n):
            if curAction != np.argmax(self.Q[state]):
                prob[curAction] = self.options.epsilon / self.env.action_space.n
            else:
                prob[curAction] = 1 - self.options.epsilon + self.options.epsilon / self.env.action_space.n
        return prob
```

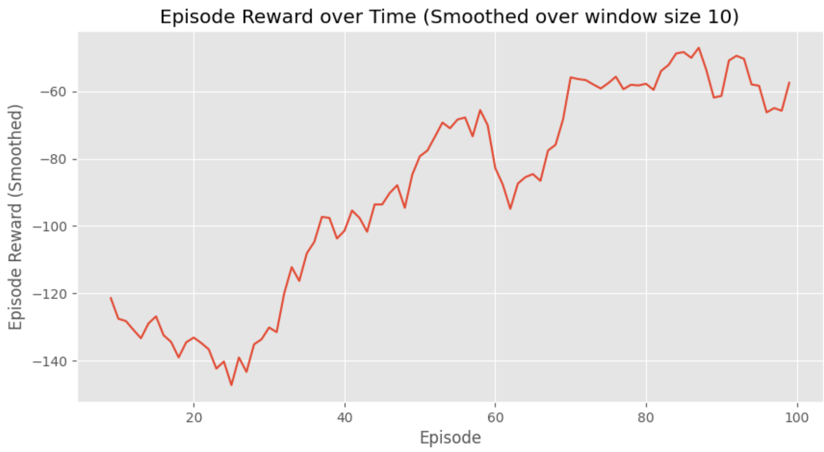


Figure 3: Plot for python run.py -s ql -d CliffWalking -e 100 -a 0.2 -g 0.9 -p 0.1

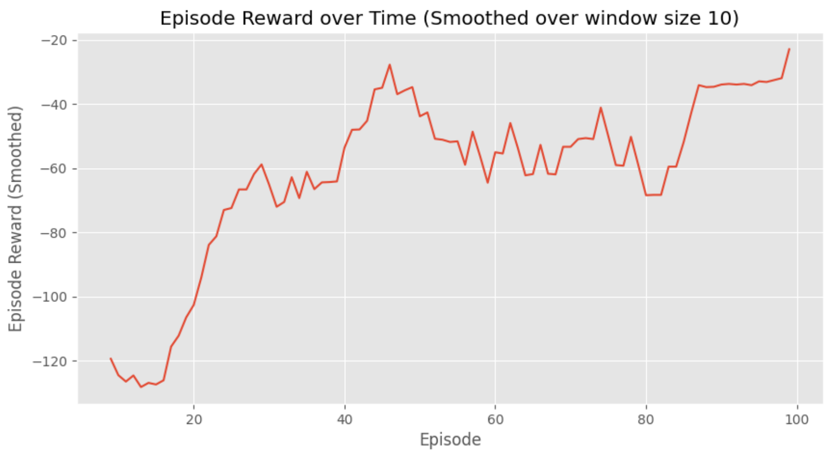


Figure 4: Plot for python run.py -s ql -d CliffWalking -e 100 -a 0.8 -g 0.5 -p 0.1

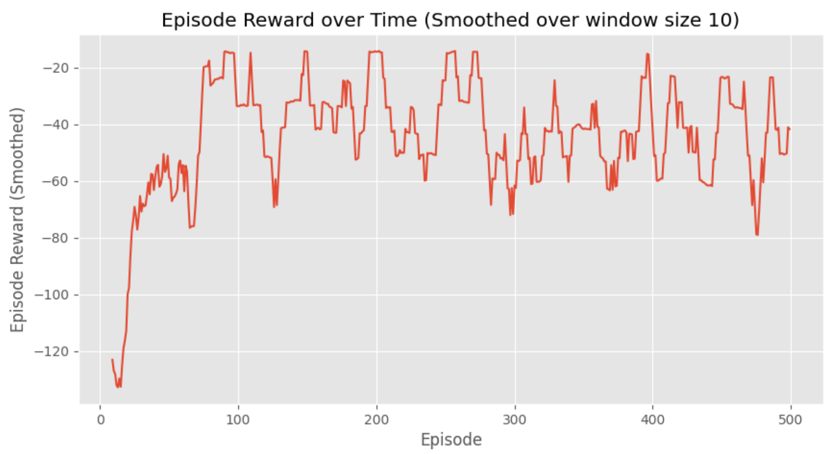
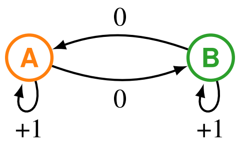


Figure 5: Plot for python run.py -s ql -d CliffWalking -e 500 -a 0.6 -g 0.8 -p 0.1

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.

	move	stay
A	0	0
B	0	0

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 $\epsilon = 0.5$.

	move	stay
A	3.988	4.999
B	3.999	4.993

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.

$$\text{Bellman Equation: } Q^*(s, a) = \sum_{s'} P(s'|s, a) \left[R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right]$$

As the transition is deterministic, $P(s'|s, a) = 1$ if $s' = s_{t+1}$ and 0 otherwise.

$$Q(A, \text{move}) = 1.0 * (0 + 0.8) = 0.8$$
$$Q(B, \text{move}) = 1.0 * (0 + 0.8) = 0.8$$
$$Q(A, \text{stay}) = 1.0 * (1 + 0.8) = 1.8$$
$$Q(B, \text{stay}) = 1.0 * (1 + 0.8) = 1.8$$

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 in the deterministic greedy behavior policy it tries to exploit rather than explore (the arg max is the best action). In the eplison greedy policy it tries to explore as it doesn't always take the argmax.

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.

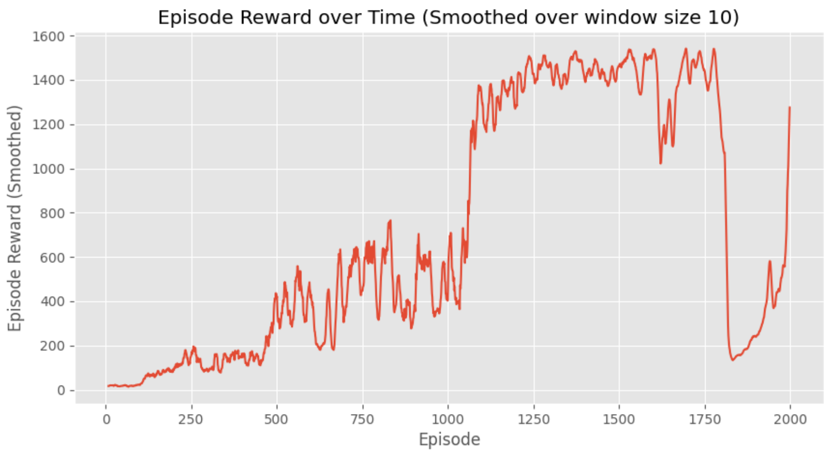


Figure 6: alpha:0.001

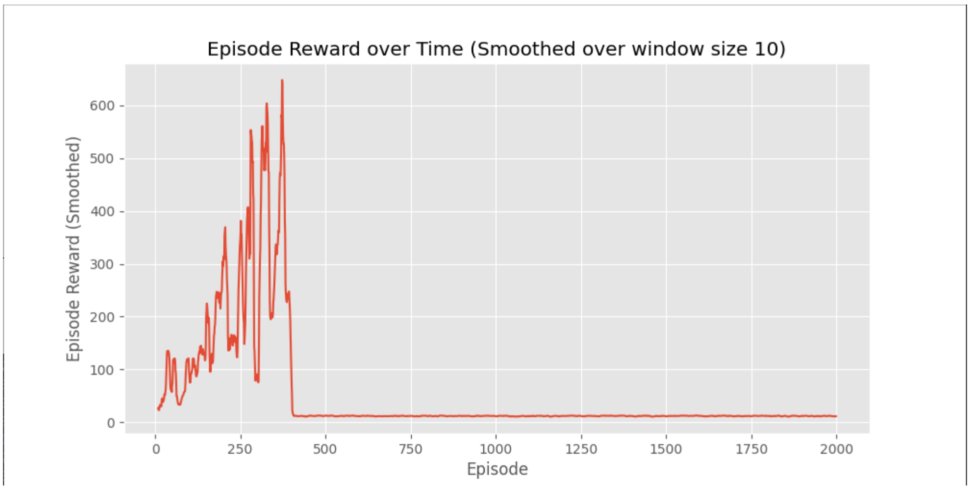


Figure 7: alpha:0.01

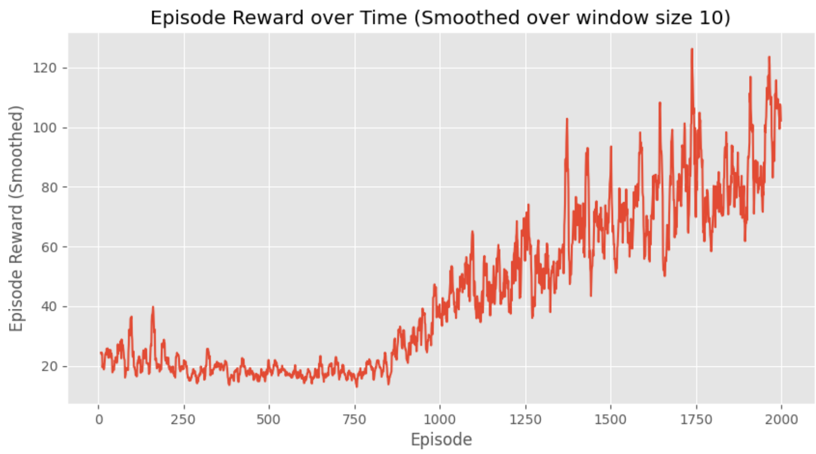


Figure 8: alpha:0.0001

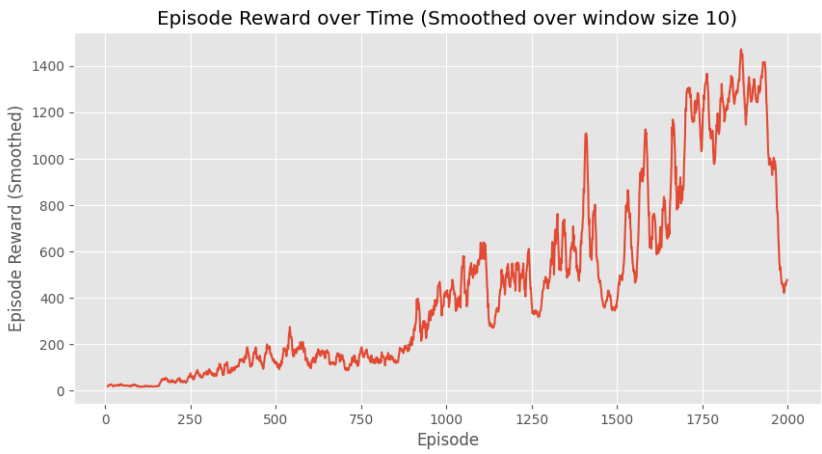


Figure 9: alpha:0.0005

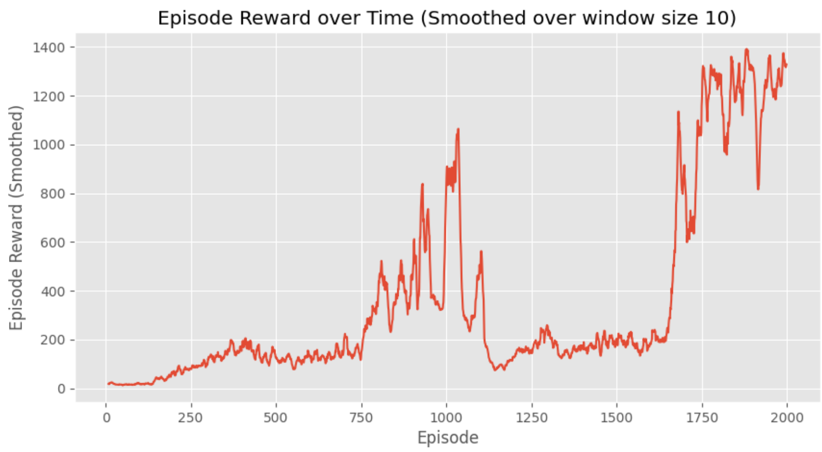


Figure 10: alpha:0.00075

I tested with 5 different alpha values. I started with 0.0001 and saw that the results were ok and used that as a base line. I tried increase the alpha to 0.01 and saw that the reward was much lower. I decreased the alpha to 0.0001 and saw that it wasn't as good as the baseline, but still better than 0.01. I then tried to use binary search in a sense to find the best alpha and narrowed it down to 0.00075 as the best alpha value as it resulted in the best reward overall.

```python
def train(self, states, actions, rewards, next_state, done):
    """
    Perform single A2C update.

    states: list of states.
    actions: list of actions taken.
    rewards: list of rewards received.
    next_state: next state received after final action.
    done: if episode ended after last action was taken.
    """

    states_tensor = torch.tensor(states, dtype=torch.float32)
    next_state_tensor = torch.tensor(next_state, dtype=torch.float32, requires_grad=


    # One-hot encoding for actions
    actions_one_hot = np.zeros([len(actions), self.env.action_space.n])
    actions_one_hot[np.arange(len(actions)), actions] = 1
    actions_one_hot = torch.tensor(actions_one_hot)

    ################################
    #   YOUR IMPLEMENTATION HERE   #
    ################################
    # Compute returns
    returns = np.zeros_like(rewards)
    # TODO: Compute bootstrapped returns for each state-action in states
    # and actions.
    G = 0 if done else self.actor_critic.value(next_state_tensor).item()
    for i in reversed(range(len(rewards))):
        G = rewards[i] + (self.options.gamma * G)
        returns[i] = G

    returns = torch.tensor(returns, dtype=torch.float32)

    values = self.actor_critic.value(states_tensor)

    # TODO: Compute advantages for each state-action pair in states and
    # actions.

    advantages = returns - values.detach()

    log_probs = torch.sum(
        self.actor_critic.log_probs(states_tensor) * actions_one_hot,
        axis=-1
    )

    # Compute actor and critic losses
    ################################
    #   YOUR IMPLEMENTATION HERE   #
    ################################
    # TODO: compute these losses.
    # Useful functions: torch.square for critic loss.

    policy_loss = -log_probs * advantages  # Negative for gradient ascent
    critic_loss = torch.square(values - returns)


    loss = policy_loss.mean() + critic_loss.mean()

    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

Figure 11: train() code