HOMEWORK 7

>>NAME HERE<<
>>ID HERE<<

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
```

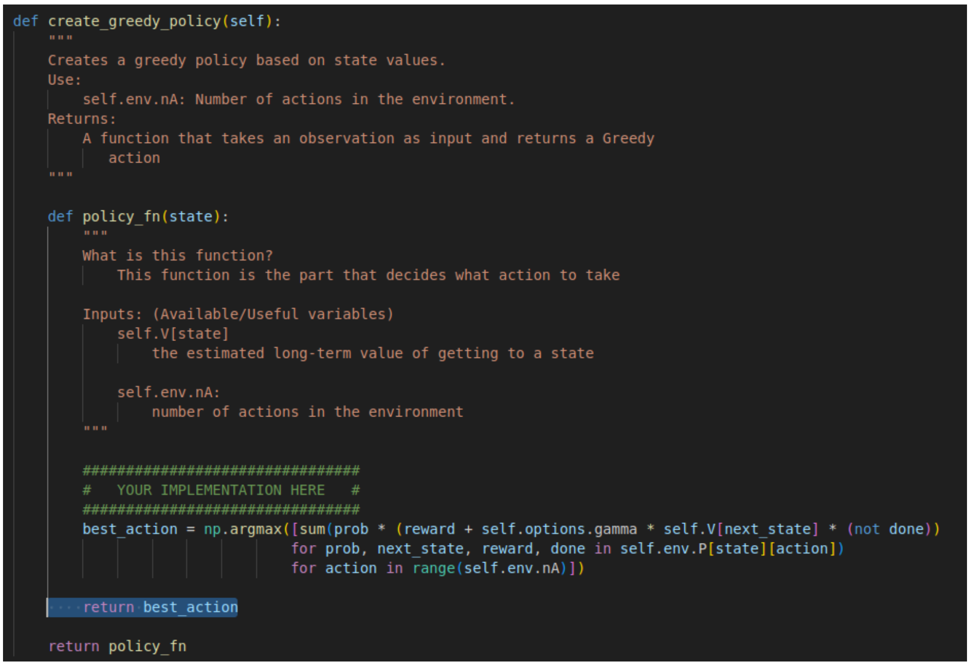
For Windows or more explanation see here: https://docs.python.org/3/tutorial/venv.html

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.

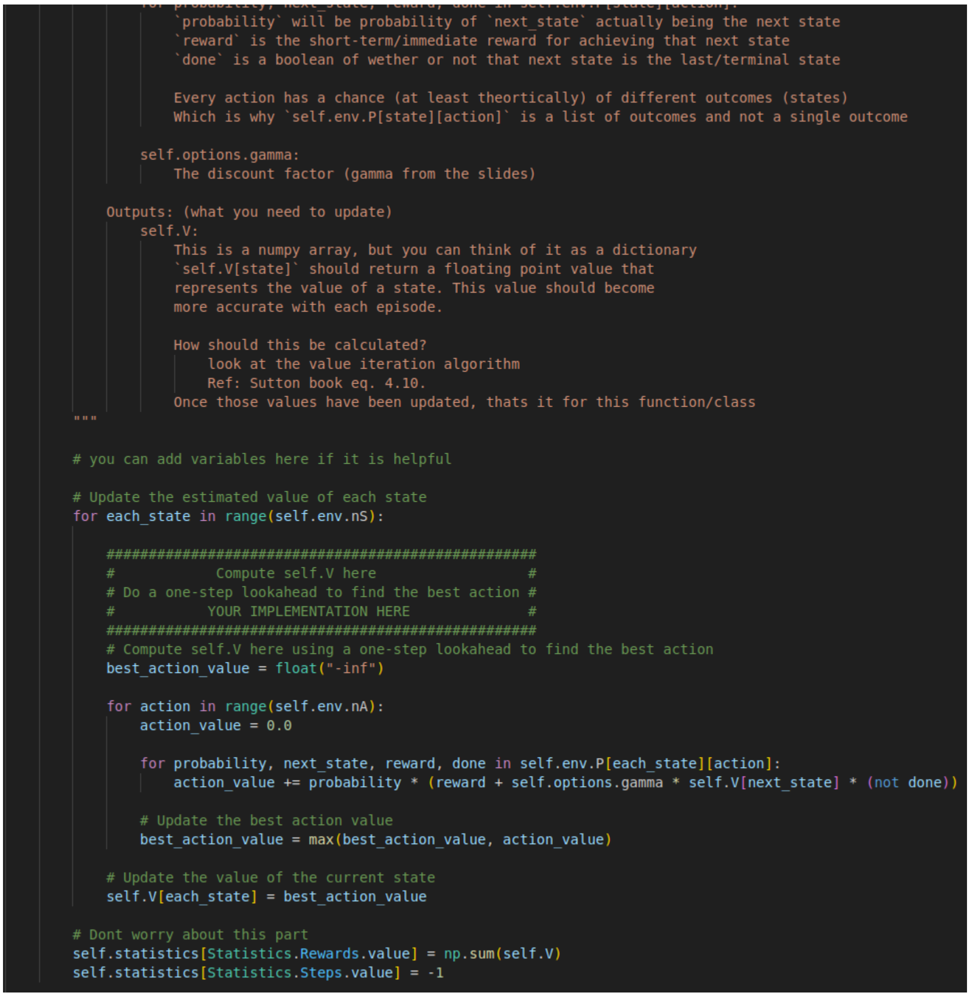
Submission [6 pts each + 10 pts for code submission]

Submit a screenshot containing your `train_episode` and `create_greedy_policy` methods (10 points). Latex code to include image



```
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



```
                for probability, next_state, reward, done in self.env.P[state][action]:
                    `probability` will be probability of `next_state` actually being the next state
                    `reward` is the short-term/immediate reward for achieving that next state
                    `done` is a boolean of wether or not that next state is the last/terminal state

                    Every action has a chance (at least theortically) of different outcomes (states)
                    Which is why `self.env.P[state][action]` is a list of outcomes and not a single outcome

                self.options.gamma:
                    The discount factor (gamma from the slides)

            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))

                # Update the best action value
                best_action_value = max(best_action_value, action_value)

            # Update the value of the current state
            self.V[each_state] = best_action_value

        # Dont worry about this part
        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

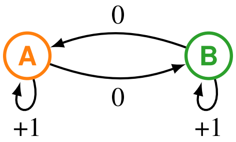
Again for this command, the expected reward is given for a correct solution. If your solution gives the same reward it doesn't guarantee correctness on the test cases.

```
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.

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.

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 $\epsilon = 0.5$.

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.

   The true optimal action-values for the MDP are derived using the Bellman equation:

   $$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(s', a')$$

   Given the MDP has two states (A and B) and two actions (move and stay), with deterministic transitions and rewards of 0 for 'move' and 1 for 'stay', we have:

   $$\begin{aligned} Q(A, \text{stay}) &= 1 + 0.8 \max(Q(A, \text{stay}), Q(A, \text{move})) \\ Q(A, \text{move}) &= 0.8 \max(Q(B, \text{stay}), Q(B, \text{move})) \\ Q(B, \text{stay}) &= 1 + 0.8 \max(Q(B, \text{stay}), Q(B, \text{move})) \\ Q(B, \text{move}) &= 0.8 \max(Q(A, \text{stay}), Q(A, \text{move})) \end{aligned}$$

   Considering the symmetry in the MDP, we simplify the equations to:

$$
\begin{aligned}
Q_s &= 1 + 0.8Q_s \\
Q_m &= 0.8Q_s
\end{aligned}
$$

Solving these equations, we find:

$$
\begin{aligned}
Q_s &= 5.0 \\
Q_m &= 4.0
\end{aligned}
$$

Thus, the true optimal action-values for the MDP are:

$$
\begin{aligned}
Q(A, \text{stay}) = Q(B, \text{stay}) &= 5.0 \\
Q(A, \text{move}) = Q(B, \text{move}) &= 4.0
\end{aligned}
$$

These values indicate that the optimal action in each state is to 'stay', which aligns with the immediate reward of 1 for staying, as opposed to moving, which yields no immediate reward.

4. (5 pts) To the extent that you obtain different solutions for each question, explain why the action-values differ.
The differences in action-values obtained from Q-learning and theoretical calculation using the Bellman equation can be attributed to the following:

   - **Exploration and Exploitation**: The ϵ-greedy policy in Q-learning involves exploration, which can lead to variations in learned action-values compared to the optimal values derived from the Bellman equation.
   - **Initial Conditions and Learning Rate**: Q-learning's outcomes depend on the initial Q-values and the learning rate (α). Different initializations and learning rates can lead to different learning trajectories.
   - **Sample-Based Learning vs. Theoretical Calculation**: Q-learning is a sample-based approach and may not fully capture the state-action space within a limited number of steps, unlike the Bellman equation which assumes perfect knowledge of the MDP.
   - **Policy Differences**: The deterministic greedy policy and the ϵ-greedy policy in Q-learning might lead to different action selections compared to the optimal policy assumed in theoretical calculations.
   - **Convergence Time**: The number of steps in Q-learning (200 in this case) might not be sufficient for convergence to the true optimal values.

   In summary, the discrepancies arise from the differences in learning mechanisms and assumptions between practical Q-learning algorithms and theoretical optimal value calculations.

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.