HOMEWORK 7

>>Hemanth Sridhar Nakshatri<<
>>nakshatri@wisc.edu, 9085807346<<

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.

The codes haven't been uploaded to GitHub since it was not asked to be uploaded. All the previous homeworks can be found in the repository, https://github.com/hemanth-nakshatri/ECE760-Machine-Learning

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).

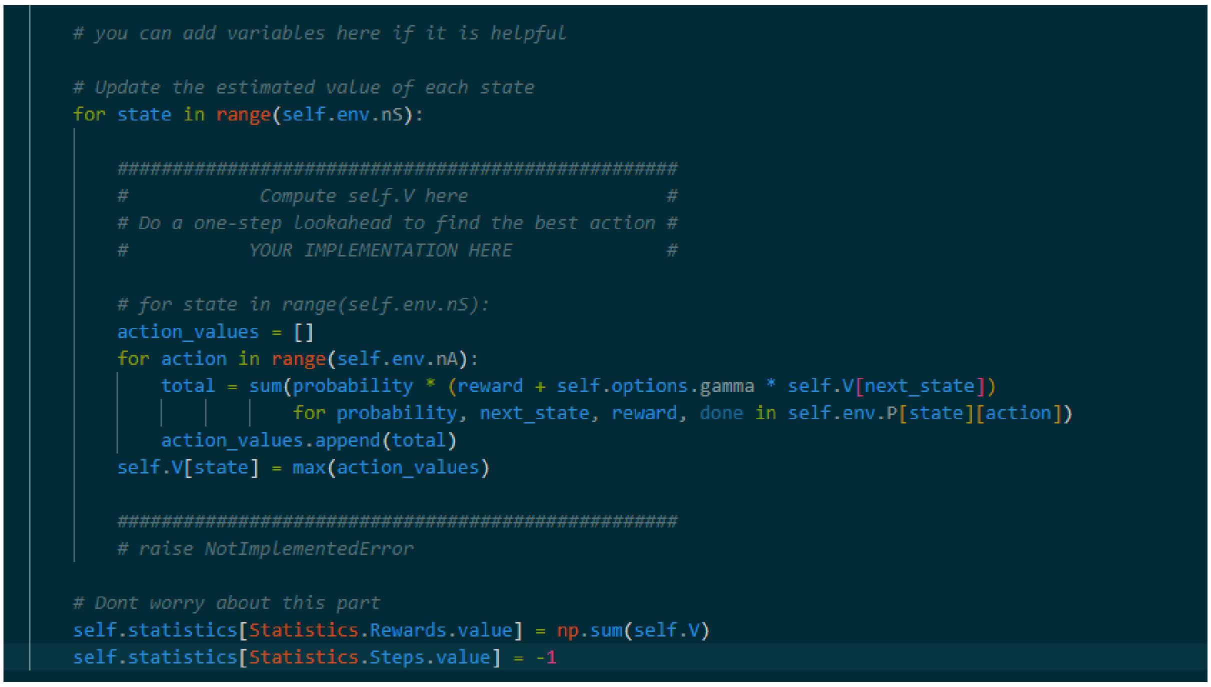
For these 5 commands. Report the episode it converges at and the reward it achieves. See examples for what we expect. An example is:

```
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.

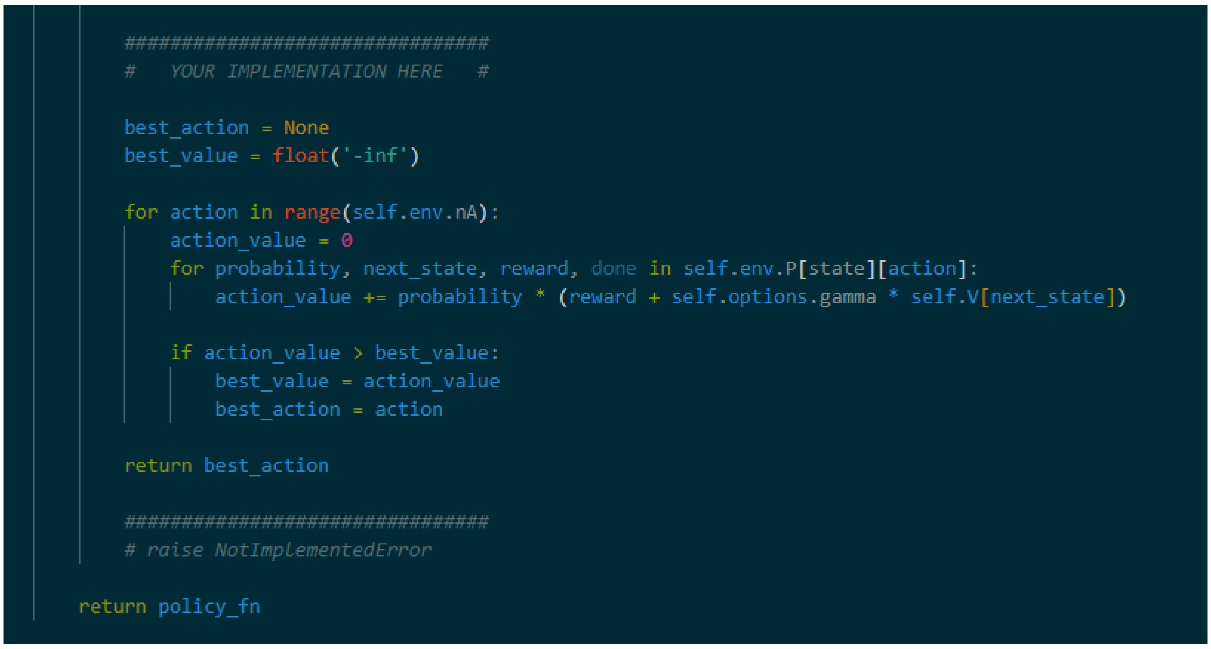
Figure. 1 and Figure. 2 show the implementation methods for the *train episode* and *create greedy policy* methods for the Value iteration method.
Submission Commands:

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 **0.63743** in **14** episodes
4. python run.py -s vi -d FrozenLake-v0 -e 500 -g 0.9
   Converges to a reward of **2.17609** in **73** episodes



```python
# you can add variables here if it is helpful

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

    ###################################################
    #            Compute self.V here                  #
    # Do a one-step lookahead to find the best action #
    #           YOUR IMPLEMENTATION HERE              #

    # for state in range(self.env.nS):
    action_values = []
    for action in range(self.env.nA):
        total = sum(probability * (reward + self.options.gamma * self.V[next_state])
                    for probability, next_state, reward, done in self.env.P[state][action])
        action_values.append(total)
    self.V[state] = max(action_values)

    ###################################################
    # raise NotImplementedError

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

Figure 1: Snapshot of code for *train episode* function for Value iteration method.



```python
    #################################
    #   YOUR IMPLEMENTATION HERE   #

    best_action = None
    best_value = float('-inf')

    for action in range(self.env.nA):
        action_value = 0
        for probability, next_state, reward, done in self.env.P[state][action]:
            action_value += probability * (reward + self.options.gamma * self.V[next_state])

        if action_value > best_value:
            best_value = action_value
            best_action = action

    return best_action

    #################################
    # raise NotImplementedError

return policy_fn
```

Figure 2: Snapshot of code for *create greedy policy* function for Value iteration method.

5. python run.py -s vi -d FrozenLake-v0 -e 500 -g 0.75
   Converges to a reward of **1.13161** in **28** 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).

Figure. 3 and Figure. 4 show the implementation methods for the *train episode* and *create greedy policy* methods for the Q Learning method.
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

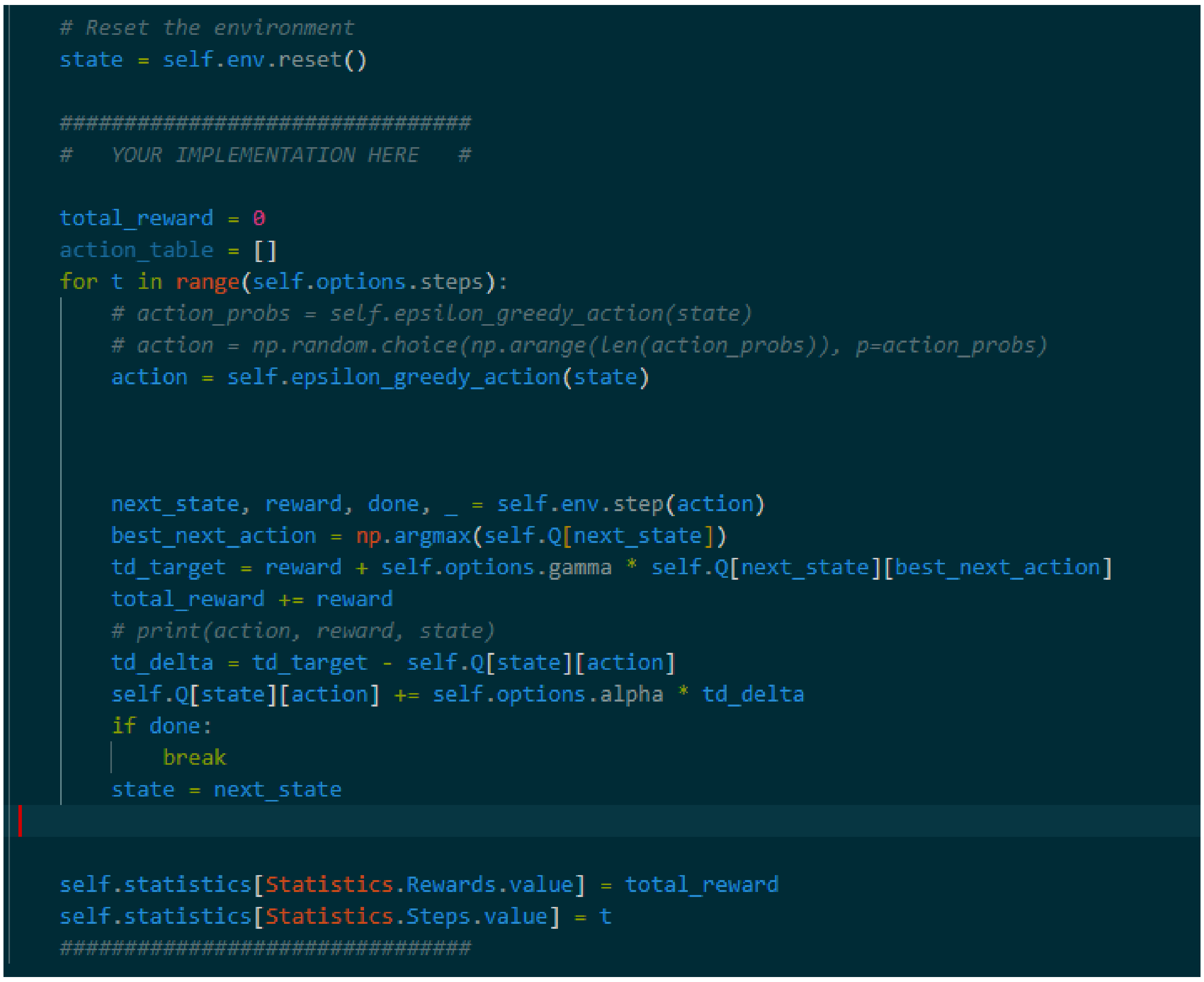
For reference, command 1 should end with a reward around -60, command 2 should end with a reward around -25 and command 3 should end with a reward around -40.
Figure. 5 and Figure. 6 show the "Episode reward over time" plots of the 3 commands for the Q Learning method.

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.

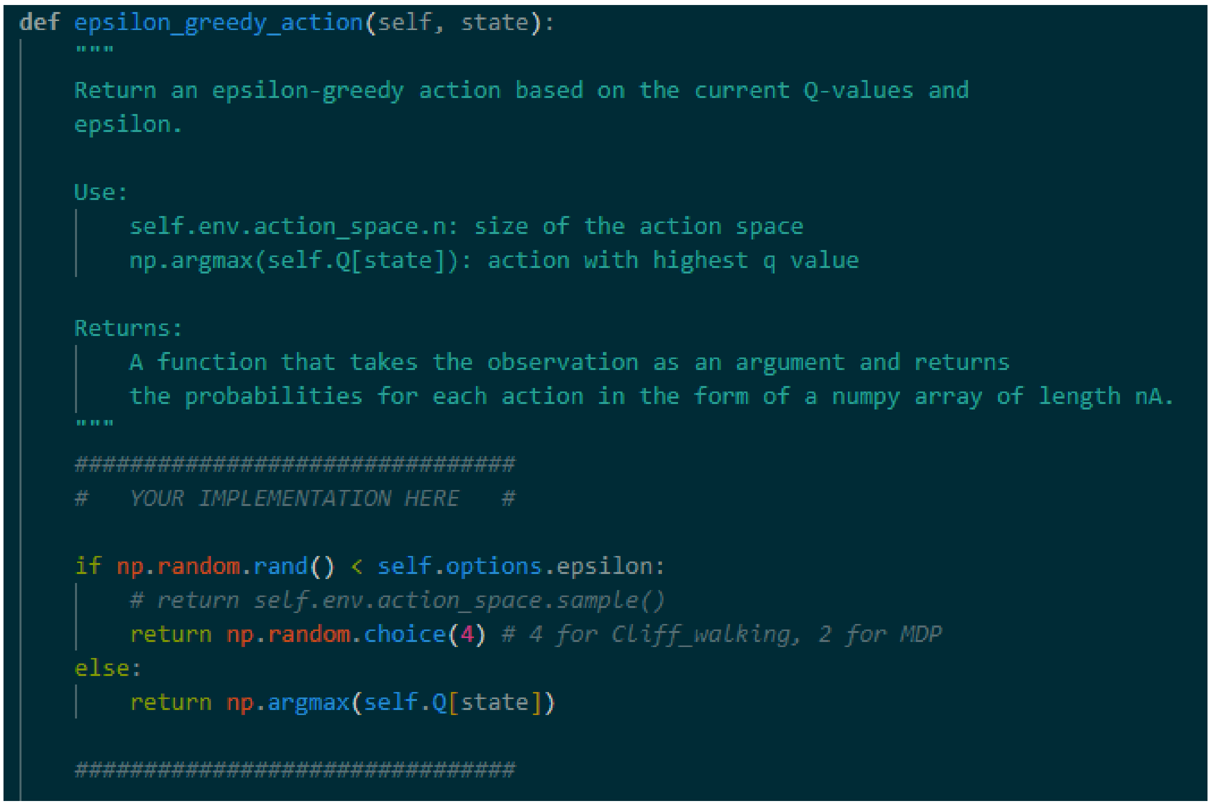
```
python run.py -s ql -d CliffWalking -e 500 -a 0.5 -g 1.0 -p 0.1
```

Achieves a best performing policy with -13 reward.



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

##################################
#   YOUR IMPLEMENTATION HERE   #

total_reward = 0
action_table = []
for t in range(self.options.steps):
    # action_probs = self.epsilon_greedy_action(state)
    # action = np.random.choice(np.arange(len(action_probs)), p=action_probs)
    action = self.epsilon_greedy_action(state)



    next_state, reward, done, _ = self.env.step(action)
    best_next_action = np.argmax(self.Q[next_state])
    td_target = reward + self.options.gamma * self.Q[next_state][best_next_action]
    total_reward += reward
    # print(action, reward, state)
    td_delta = td_target - self.Q[state][action]
    self.Q[state][action] += self.options.alpha * td_delta
    if done:
        break
    state = next_state


self.statistics[Statistics.Rewards.value] = total_reward
self.statistics[Statistics.Steps.value] = t
##################################
```

Figure 3: Snapshot of code for *train episode* function for Q Learning method.



```
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   #

    if np.random.rand() < self.options.epsilon:
        # return self.env.action_space.sample()
        return np.random.choice(4) # 4 for Cliff_walking, 2 for MDP
    else:
        return np.argmax(self.Q[state])

    ##################################
```

Figure 4: Snapshot of code for *epsilon greedy action* function for Q Learning method.

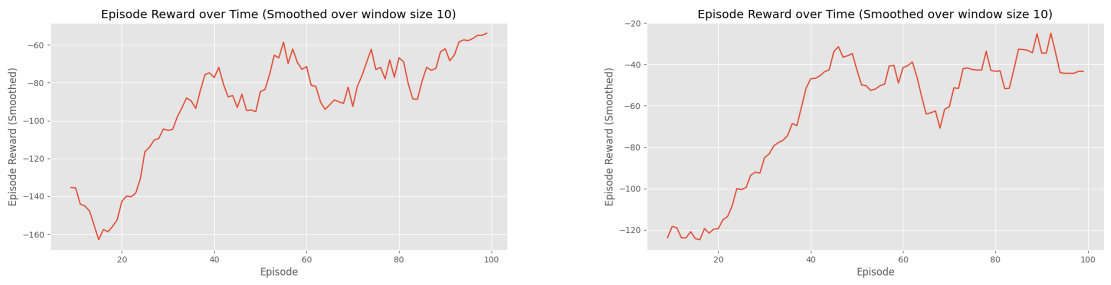


Figure 5: *Reward over time* plot for the first two commands for Q Learning (Quesion 3).

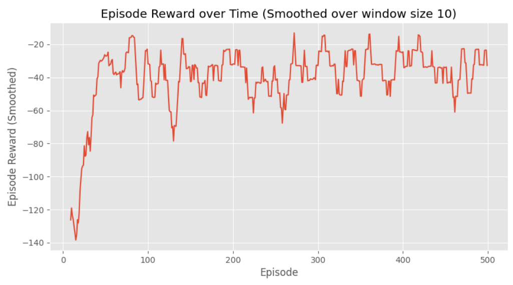


Figure 6: *Reward over time* plot for the last command for Q Learning (Quesion 3).

State/Action	Move (Action 0)	Stay (Action 1)
A	0.0	0.0
B	0.0	0.0

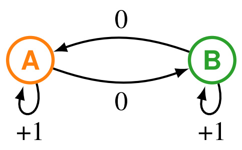
Table 1: Action value table for Q4.1 (Choose the best action) and break tie by prefering move.

State/Action	Move (Action 0)	Stay (Action 1)
A	3.99272	4.99914
B	3.99912	4.99524

Table 2: Action value table for Q4.2 (Choose the best action with $(1-\epsilon)$ probability) and break tie arbitrarily.

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.

   Table 1 shows the action-value table at the end of 200 steps for the above greedy policy. Since the actions begin at 0 for both stay and move, the tie breaker policy always forces to switch states resulting in 0 rewards. Thus the state keeps changing at every step.

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 $\epsilon = 0.5$.

   Table 2 shows the action-value table at the end of 200 steps for the above greedy policy. The action to "Stay" is rewarded more and the "Move" action has a bit lesser value since it promotes to "Stay" in the next step in case the current action is "Move".

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.

   Bellman's optimality equation for Q-values is given by,
   $Q^*(s,a) = \mathbb{E}[R_{t+1} + \gamma\ max_{a'} Q * (S_{t+1},a')|S_t = s, A_t = a]$

   Since the actions and rewards are deterministic, the above equation can be simplified to:
   $Q^*(s,a) = r(s,a) + \gamma\ max_{a'} Q^*(s',a')$

   To calculate the True Optimal values:

   (a) **For state A:**

State/Action	Move (Action 0)	Stay (Action 1)
A	4	5
B	4	5

Table 3: Action value table for Q4.3. True optimal values calculation by hand

- **Move**: $r = 0 \cdot Q^*(A, move) = 0 + 0.8 \cdot max(Q^*(B, move), Q^*(B, stay))$
- **Stay**: $r = 1 \cdot Q^*(A, stay) = 1 + 0.8 \cdot max(Q^*(A, move), Q^*(A, stay))$

(b) Similarly **for state B:**

- **Move**: $r = 0 \cdot Q^*(B, move) = 0 + 0.8 \cdot max(Q^*(A, move), Q^*(A, stay))$
- **Stay**: $r = 1 \cdot Q^*(B, stay) = 1 + 0.8 \cdot max(Q^*(B, move), Q^*(B, stay))$

Since the reward is 1 for staying and 0 for moving, $max(Q^*(B, move), Q^*(B, stay))$ and $max(Q^*(A, move), Q^*(A, stay))$ is equal to $Q^*(B, stay)$ and $Q^*(A, stay)$ respectively.

Thus $Q^*(A, stay) = 1 + 0.8 * Q^*(A, stay)$
$\mathbf{Q^*(A, stay) = \dfrac{1}{1 - 0.8} = 5}$
Similarly, $\mathbf{Q^*(B, stay) = \dfrac{1}{1 - 0.8} = 5}$

Substituting these values will give us:
$\mathbf{Q^*(A, move) = 0 + 0.8 * Q^*(B, stay) = 4}$
$\mathbf{Q^*(B, move) = 0 + 0.8 * Q^*(A, stay) = 4}$

Table. 3 shows the True optimal action values calculations by hand.

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

The difference in the actual action values vs the true optimal values could be due to different reasons.

- **Finite number of steps:** In this case, we have considered only one episode and 200 steps. Ideally Q Learning method converges to true optimal values with infinite steps. Thus the finite steps could be a reason.
- **Learning rate α:** If α is large, it might overshoot the optimal value. But a small α could make the descent too slow. From the action values, it can be seen the actual action values are really close to true optimal value indicating that the α might be a bit too big to reach further closer to the optimal value.
- **Epsilon ϵ :** ϵ value can affect the action selection policy which will in turn affect the convergence.
- **Precision:** The numerical precision issues will probably add a minor discrepancy as well.

Thus, the difference in the action values is due to a combination of the above mentioned factors.

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.

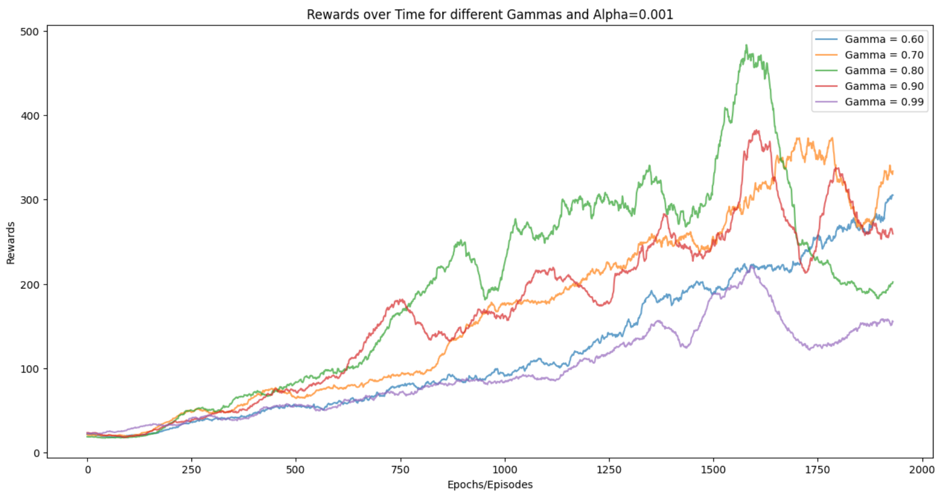


Figure 7: *Reward over time* plot for A2C with constant learning rate $\alpha = 0.001$

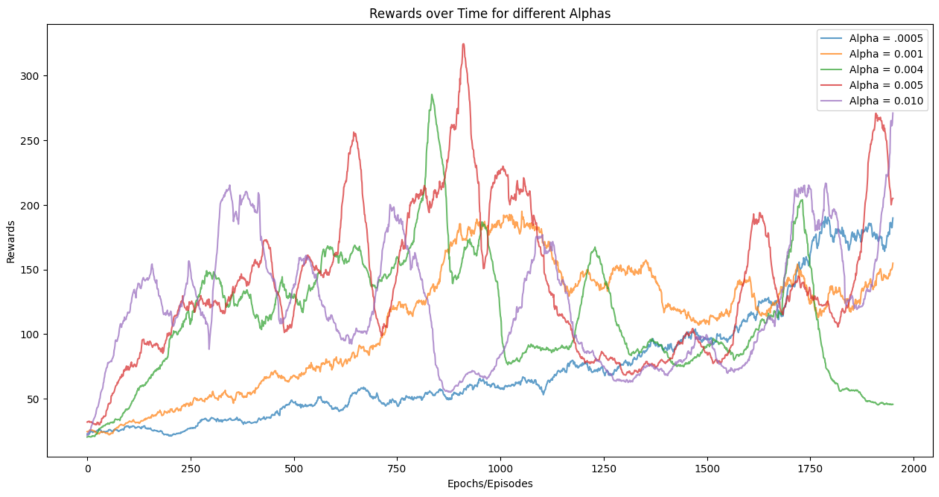


Figure 8: *Reward over time* plot for A2C with constant discount factor γ

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 *train* function for A2C algorithm has been implemented and the results are shown.

- Figure. 7 shows that the rewards are lower if the discount factor is too high or too low, meaning there needs to be a balance between the importance of the current and the future rewards.
- Figure. 8 shows that higher learning rate results in faster increase in rewards over time. But it is also visible that the algorithm is unstable and the rewards plummet if the learning rate is too high.
- Thus there needs to be a balance between the hyperparameters α and γ which could be obtained via a ***hyperparameter search***.