HOMEWORK 7

>>Martin Diges<<
>>9080689699<<

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/missingnoglitch0/cs760/tree/main/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
# you can add variables here if it is helpful
policy = self.create_greedy_policy()

# Update the estimated value of each state
V_new = np.ndarray(self.env.nS)
for each_state in range(self.env.nS):
    s = each_state
    a_opt = policy(s)
    v_val = self.q(s, a_opt)
    V_new[s] = v_val

self.V = V_new # only change values of V once VI has happened. This way all calculations
# are done with respect to a static self.V

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

Figure 1: train_episode()

```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
        """

        num_actions = self.env.nA
        a_values = np.ndarray(num_actions)
        for a in range(num_actions):
            a_values[a] = self.q(state, a)
        return np.argmax(a_values)

    return policy_fn

def q(self, s, a):
    v_a = 0
    for p_s_next, s_next, r, done in self.env.P[s][a]:
        v_a += p_s_next * (r + self.options.gamma * self.V[s_next])
    return v_a
```

Figure 2: create_greedy_policy() and auxiliary function q()

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
   (please see Piazza @336 for why the values below may differ from expected)
   Converges to a reward of 0.6374 in 14 episodes.

4. python run.py -s vi -d FrozenLake-v0 -e 500 -g 0.9
   Converges to a reward of 2.1760 in 72 episodes.

5. python run.py -s vi -d FrozenLake-v0 -e 500 -g 0.75
   Converges to a reward of 1.1316 in 34 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).

```
policy = self.create_greedy_policy()
alpha = self.options.alpha
gamma = self.options.gamma

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

for i in range(self.options.steps):
    a_opt = self.epsilon_greedy_action(s)
    s_next, r, done, _ = self.step(a_opt)

    a_prime_opt = policy(s_next)
    self.Q[s][a_opt] = (1-alpha)*(self.Q[s][a_opt]) + (alpha)*(r + gamma*self.Q[s_next][a_prime_opt])
    s = s_next

    if done:
        return # exit if episode reaches terminal
```

Figure 3: train_episode()

```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.
    """

    epsilon = self.options.epsilon
    rand_float = np.random.rand()
    if rand_float <= epsilon:
        # print(f'epsilon={epsilon}, rand_float={rand_float}')
        return np.random.choice(self.env.action_space.n)

    policy = self.create_greedy_policy()

    return policy(state)
```

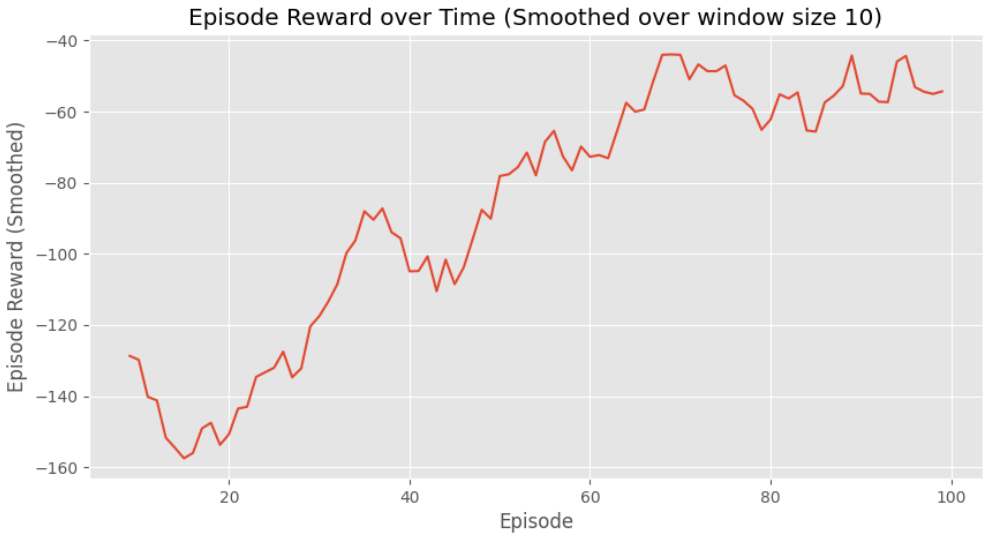
Figure 4: make_epsilon_greedy_policy()

```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
```

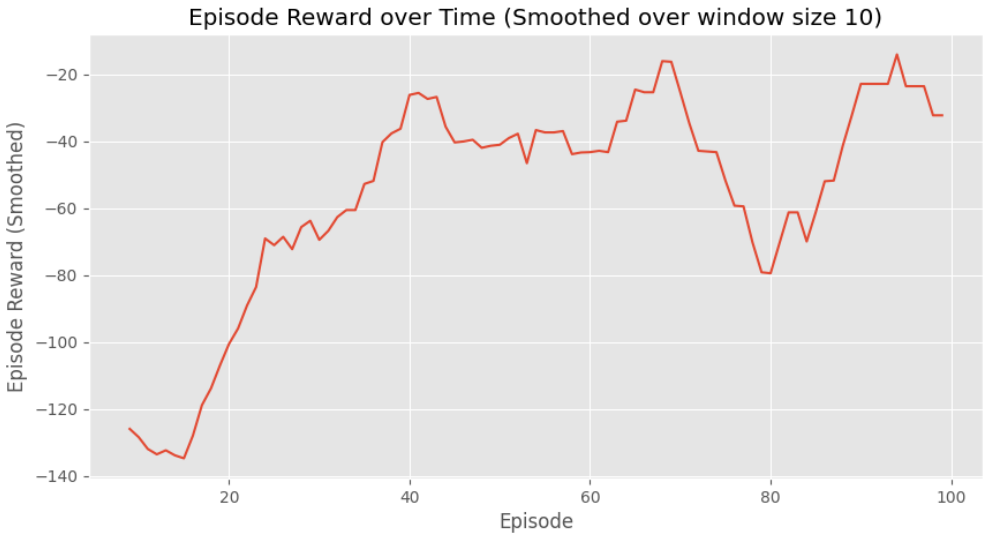
Figure 5: create_greedy_policy()

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

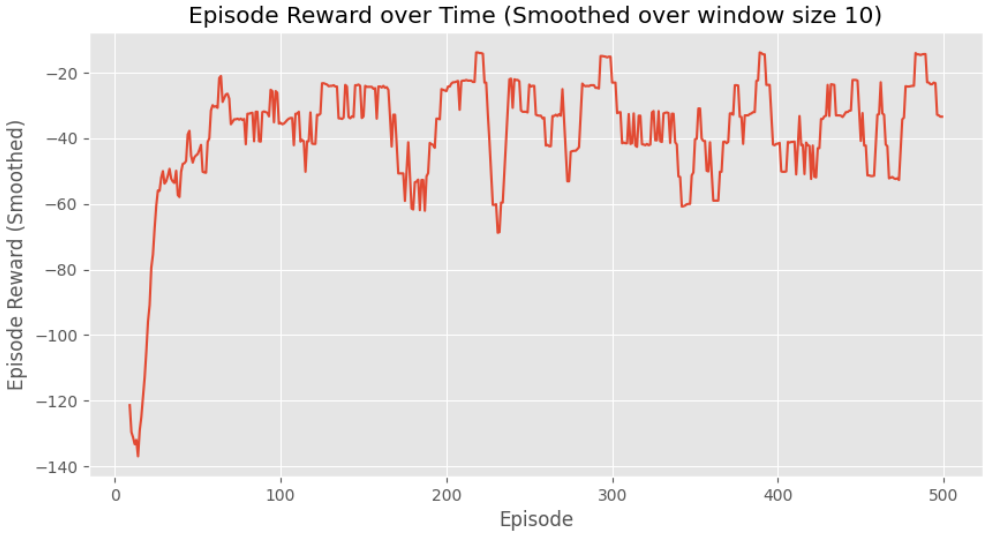
1. python run.py -s ql -d CliffWalking -e 100 -a 0.2 -g 0.9 -p 0.1



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

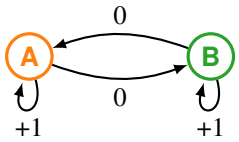
Again for this command, the expected reward is given for a correct solution. If your solution gives the same reward it doesn't guarantee correctness on the test cases.

```
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.
   {'A': {'STAY': 0, 'MOVE': 0.0}, 'B': {'STAY': 0, 'MOVE': 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 $\epsilon = 0.5$.
   {'A': {'STAY': 4.999975828096329, 'MOVE': 3.687284964530014}, 'B': {'STAY': 4.700927716064728, 'MOVE': 3.999969476317586}}

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.

   From the parameters supplied earlier, we have

   $$Q(s_t, a_t) = (0.5)Q(s_t, a_t) + (0.5) \left(r_t + 0.8 \max_{a'} Q(s_{t+1}, a') \right)$$

   $$Q(s_t, a_t) = \left(r_t + 0.8 \max_{a'} Q(s_{t+1}, a') \right)$$

   $$Q(s_t, a_t) = (r_t + 0.8 \max(Q(s_{t+1}, STAY), Q(s_{t+1}, MOVE)))$$

   $$Q(s_t, STAY) = (1 + 0.8 \max(Q(s_{t+1}, STAY), Q(s_{t+1}, MOVE)))$$

   $$Q(s_t, MOVE) = (0 + 0.8 \max(Q(s_{t+1}, STAY), Q(s_{t+1}, MOVE)))$$

   It follows that $Q(s_t, STAY) > Q(s_t, MOVE)$. Since STAY is the max valued action, we have that

   $$Q(s_t, STAY) = 1 + 0.8Q(s_{t+1}, STAY)$$

   $$Q(s_t, STAY) * (1 - 0.8) = 1$$

   $$Q(s_t, STAY) = \frac{1}{1 - 0.8} = \frac{1}{0.2} = 5$$

   and

   $$Q(s_t, MOVE) = 0 + 0.8Q(s_{t+1}, STAY) = 0.8(5) = 4$$

4. (5 pts) To the extent that you obtain different solutions for each question, explain why the action-values differ.
   For 1., We see that the action-value table retains its starting default values of 0 for all entries. This is because we explore deterministically and prefer to move when entires for stay and move are equal. Hence, we move

and have a reward of 0. The updated action-values are always 0, so we keep moving and setting action-values to 0.
For 2., the ϵ-greedy policy makes it more likely for random actions to be chosen and thus explored. This means the STAY action is sometimes chosen, leading to a > 0 action-value for each of our states. We thereby obtain nonzero action values.
In 3., we find the steady-state value for our ϵ-greedy behavior, which is the set of values where applying our update rule would not lead to any changes in action-values.

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.