UMass CMPSCI 383 (AI) HW5: Chapters 16, 17 & 21

YOUR NAME HERE

Assigned: Nov 9 2017; Due: Nov 29 2017 @ 11:55 PM EST

Abstract

Submit a (.zip) file to both Moodle and Gradescope containing your latex (.tex) file and rendered pdf. All written HW responses should be done in latex (use sharelatex.com or overleaf.com). All code should be written in Python and should run on the edlab machines ([yourusername]@elnux[1,2,...].cs.umass.edu).

For this HW, you be learning the Utility (Value) function and optimal policy for the Grid World domain from your text. You will learn U^* and π^* both when you have the transition probabilities and reward functions (MDP) and when you do not have them (RL). For many of the questions, you should check your answers against the figures in the book.

We have provided you with the Grid World domain. The `GridWorld_MDP` class includes the transition probability function and reward function whereas the `GridWorld` class does not. Do NOT edit these classes. We will test your code assuming these were left unchanged.

In the Grid World, your agent receives the reward for state s, $R(s)$, after the agent executes an action in that state. For example, assume the agent is battery powered. There is a -0.04 reward in each state to represent the cost associated with draining its battery. Let's say the agent starts in state [2,0] at time t=0 with full battery. Now, let's say the agent decides to move up and arrives at [1,0] at time t=1. The agent receives a reward of -0.04 indicating it has lost some battery power —we can interpret this as the agent paying $0.04 to recharge its battery.

Once the agent reaches one of the terminal states, any action in one of those states will take the agent to the absorbing state. At this point, the "epsiode" has ended, and the agent will stay in this state forever. You should use the `reset*` methods to return the agent to the start state or a random tile.

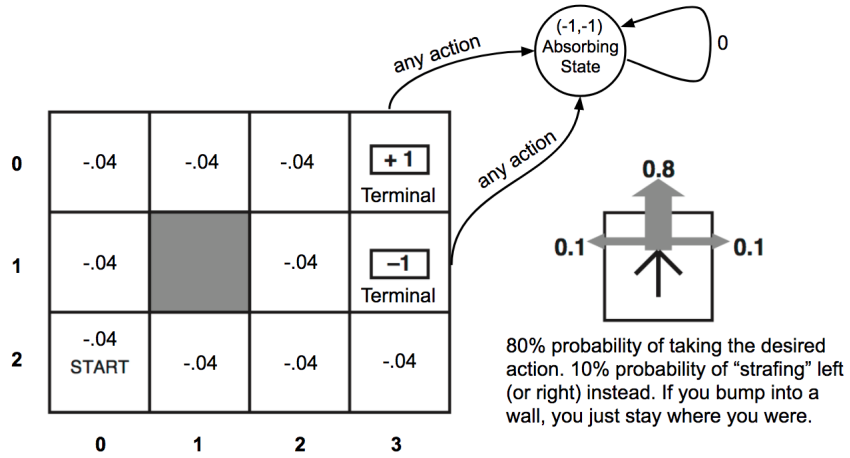


Figure 1: Gridworld MDP

You are encouraged to use the numpy package for this assignment. The template code we gave you has already imported numpy. A few methods you will find useful are:

1. `np.array` converts a list or nested lists of any depth into an array that you can easily index and slice, e.g., `P = np.array([[0.3,0.7],[0.4,0.6]])`, `P[0,0]` returns `0.3`, `P[:,1]` returns `array([0.7,0.6])`. See numpy docs for more info.
2. `np.eye(d)` creates an identity matrix of size d.
3. `np.linalg.pinv(A)` takes the pseudoinverse of the matrix (2-d numpy array) A.
4. `np.dot(A,b)` performs the matrix multiplication of A and b (b can be a 1-d numpy array).

1 Solving MDPs (35 pts)

Here, you will examine policy iteration using the `GridWorld_MDP` class.

1. Work through the first iteration of policy iteration with an initial policy that goes up in every state, i.e., $\pi(s) = 0$. Show the resulting U_1 and π_1. Show your work. (10 pts)

   Mapping X_i to $U_1(x, y)$ where i = 4x + y

$$\vec{A} = \begin{bmatrix} -.1 & .1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ .1 & -.2 & .1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & .1 & -.2 & .1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ .8 & 0 & 0 & 0 & -.8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & .8 & 0 & 0 & 0 & -.9 & .1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & .8 & 0 & 0 & 0 & -.9 & .1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .1 & -.2 & .1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & .8 & 0 & 0 & .1 & 0 & .1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & .8 & 0 & 0 & .1 & -.9 \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} .04 \\ .04 \\ .04 \\ -1 \\ .04 \\ 0 \\ .04 \\ 1 \\ .04 \\ .04 \\ .04 \\ .04 \end{bmatrix}, \quad \vec{x} = \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \\ X_8 \\ X_9 \\ X_{10} \\ X_{11} \end{bmatrix}$$

   Solved using python code solving for A$\vec{x} = \vec{b}$ with $\vec{x} = A^{-1}\ \vec{b}$ and performing a single iteration of policy update:

   Call $my\_hw5.policy\_iteration\_modified(mdp)$ code output:
   U_1:
   (0, 0) : -1.4000000000000021
   (0, 1) : -1.0000000000000022
   (0, 2) : -0.2000000000000014
   (0, 3) : 1.0
   (1, 0) : -1.4500000000000024
   (1, 2) : -0.3333333333333344
   (1, 3) : -1.0
   (2, 0) : -1.4662011173184382
   (2, 1) : -1.1958100558659235
   (2, 2) : -0.5254189944134089
   (2, 3) : -0.9917132216014899

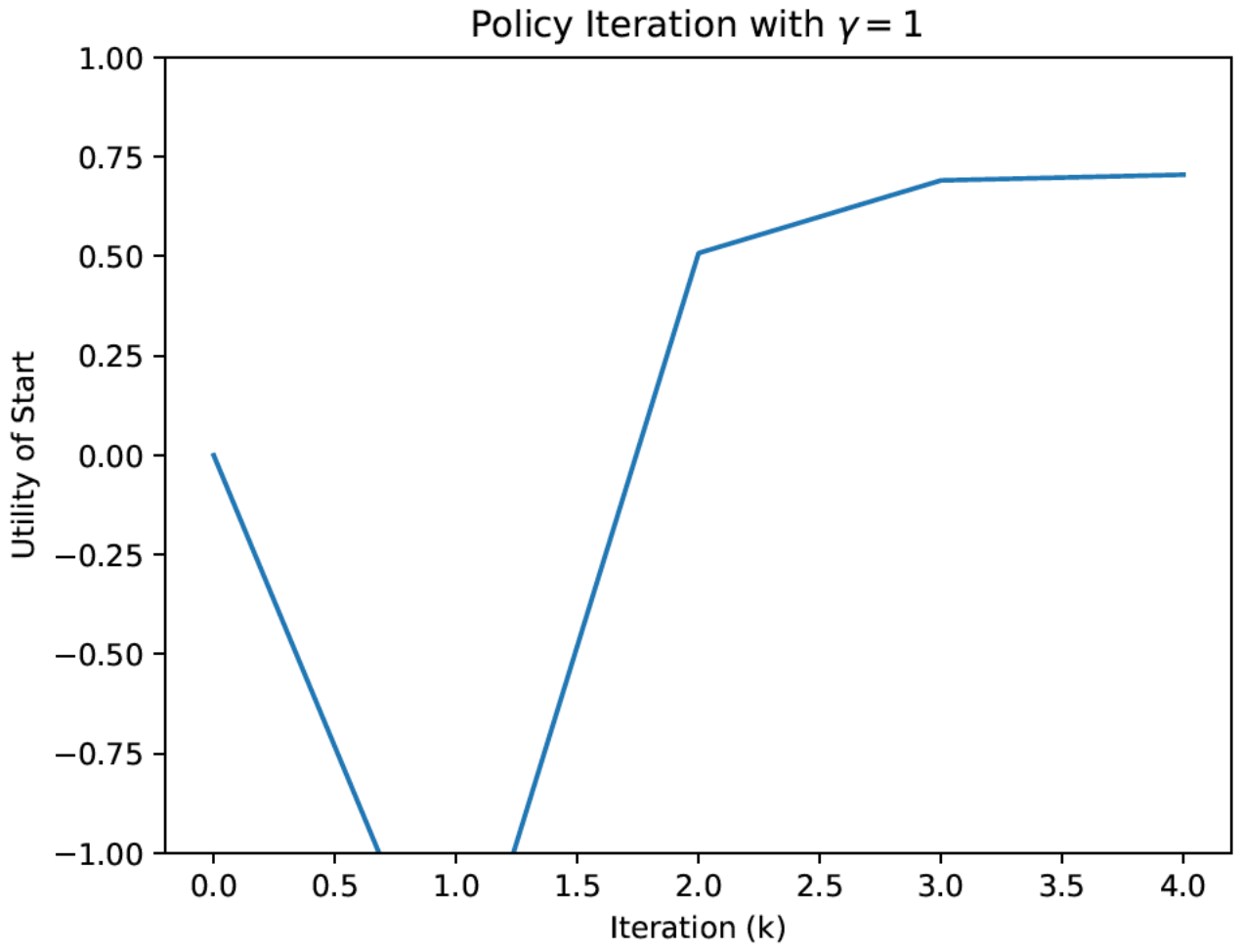
   π_1 = ['R', 'R', 'R', 'U', 'U', 'U', 'U', 'R', 'R', 'U', 'L']

2. Implement policy iteration and have it return U^* and π^*. Display them both here in table form. (15 pts for correct code)
   *Note: Although we provide you with the transition probability for going from a terminal state to an absorbing state, do not include the absorbing state or any probabilities involving it in your transition probability matrix when you create it.
   Call my_hw5.policy_iteration(x, iters=5):
   U^* = [0.81155822, 0.86780822, 0.91780822, 1. , 0.76155822, 0.66027397, -1. , 0.70530822, 0.65530822, 0.61141553, 0.38792491]
   π^* = [3, 3, 3, 0, 0, 0, 0, 0, 2, 2, 2]

Table 1: $\pi*$ and U^*

Tile	U^*	π^*
(0, 0)	0.81155822	3
(0, 1)	0.86780822	3
(0, 2)	0.91780822	3
(0, 3)	1	0
(1, 0)	0.76155822	0
(1, 2)	0.66027397	0
(1, 3)	-1	0
(2, 0)	0.70530822	0
(2, 1)	0.65530822	2
(2, 2)	0.61141553	2
(2, 3)	0.38792491	2

3. Plot $U_k(start)$ vs iterations, k, and show it is approaching $U^*(start)$. (5 pts)



The value of $U_k(Start)$ is clearly approaching $U^*(start) = .705$. The plot shows utility approaching this number.

4. We will test your code on a grid world that uses different transition probabilities to make sure your implementation is robust. (5 pts)

2 Passive RL (30 pts)

Here, you will examine TD-learning using the `GridWorld` class.

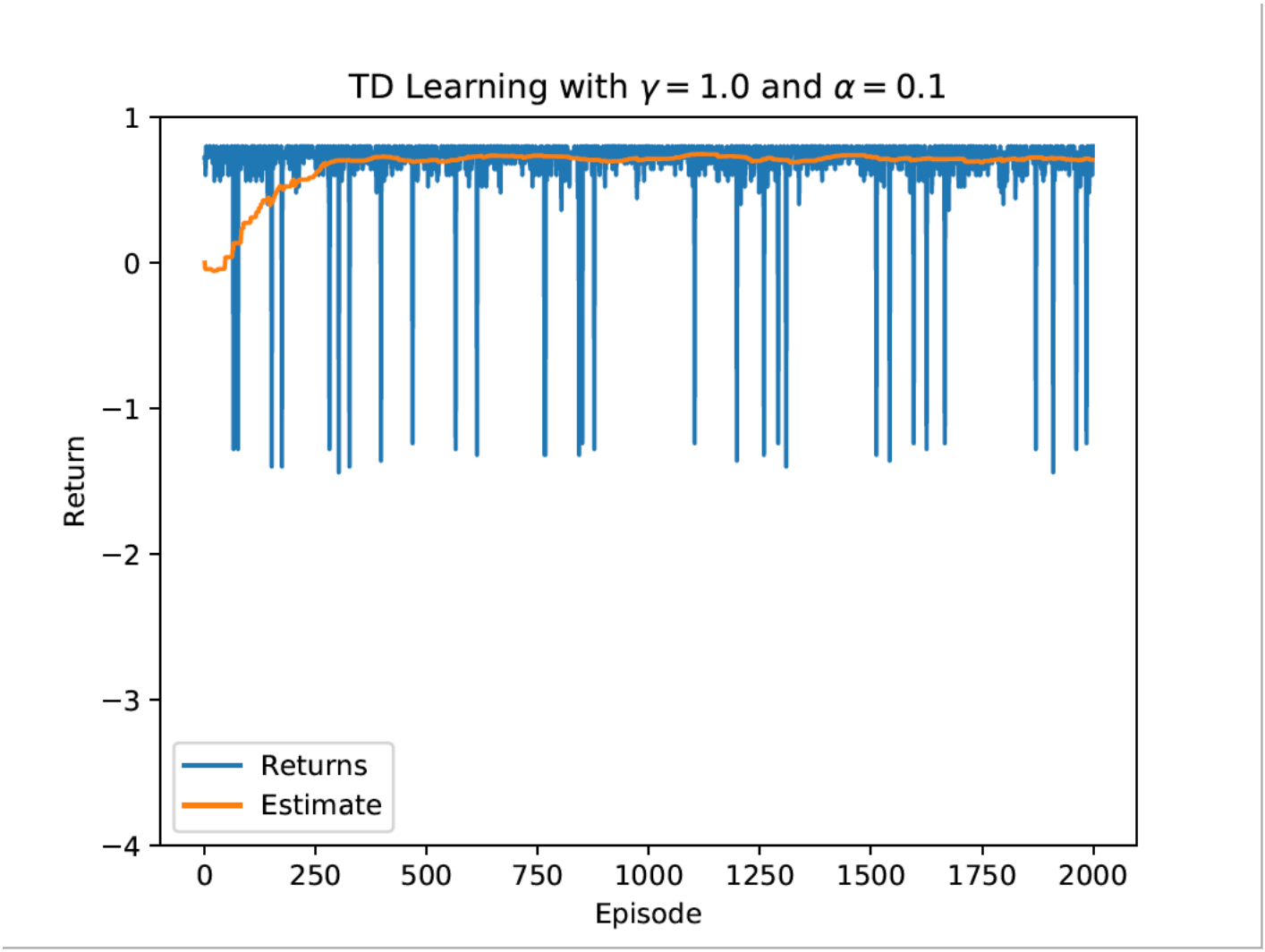
1. Work through the first iteration of TD-learning for the s-s' transition, $[2, 0] - [2, 1]$. Use learning rate $\alpha = 1$ and discount factor $\gamma = 1$. Assume $U_0([2, 0]) = 0$ and $U_0([2, 1]) = 1$. Show your work. (5 pts)

$$
\begin{aligned}
s &= (2,0) \\
R(s) &= -.04 \\
s' &= (2,1) \\
U_0(s) &= 0 \\
U_0(s') &= 1 \\
U_1(s) &= U_0(s) + (R(s) + U_0(s') - U_0(s)) \\
U_1(s) &= 0 + (-.04 + 1 - 0) \\
&= .96
\end{aligned}
$$

2. Implement TD-learning and have it return U^*. Display U^* here. (10 pts for correct code)

$U^* =$
(0, 0) : 0.7911583036093355
(0, 1) : 0.8483430201340477
(0, 2) : 0.9012207688922251
(0, 3) : 1.0
(1, 0) : 0.7431704979678379
(1, 2) : 0.4804616035947982
(1, 3) : -1.0
(2, 0) : 0.6200804788198689
(2, 1) : 0.38174009921436797
(2, 2) : 0.0
(2, 3) : 0.0

3. Plot the $U_k(start)$ at the start of each episode (not every U_k, there are many U_k's per episode). It should be approaching $U^*(start)$. The plot you make should be with the following parameters: $\gamma = 1$, $\alpha = 0.01$, number of episodes is 2000. (10 pts).



4. We will test your code on a grid world that uses different transition probabilities to make sure your implementation is robust. (5 pts)

3 Active RL (35 pts)

Here, you will examine Q-learning using the `GridWorld` class.

1. Work through the first iteration of Q-learning for the s-a-s' transition, $[2,0] - Up - [1,0]$. Use learning rate $\alpha = 1$ and discount factor $\gamma = 1$. Assume $Q_0([2,0],\cdot) = [.5,-.1,-.1,.2]$ and $Q_0([1,0],\cdot) = [-.1,-.1,.4,-.2]$. Show the resulting

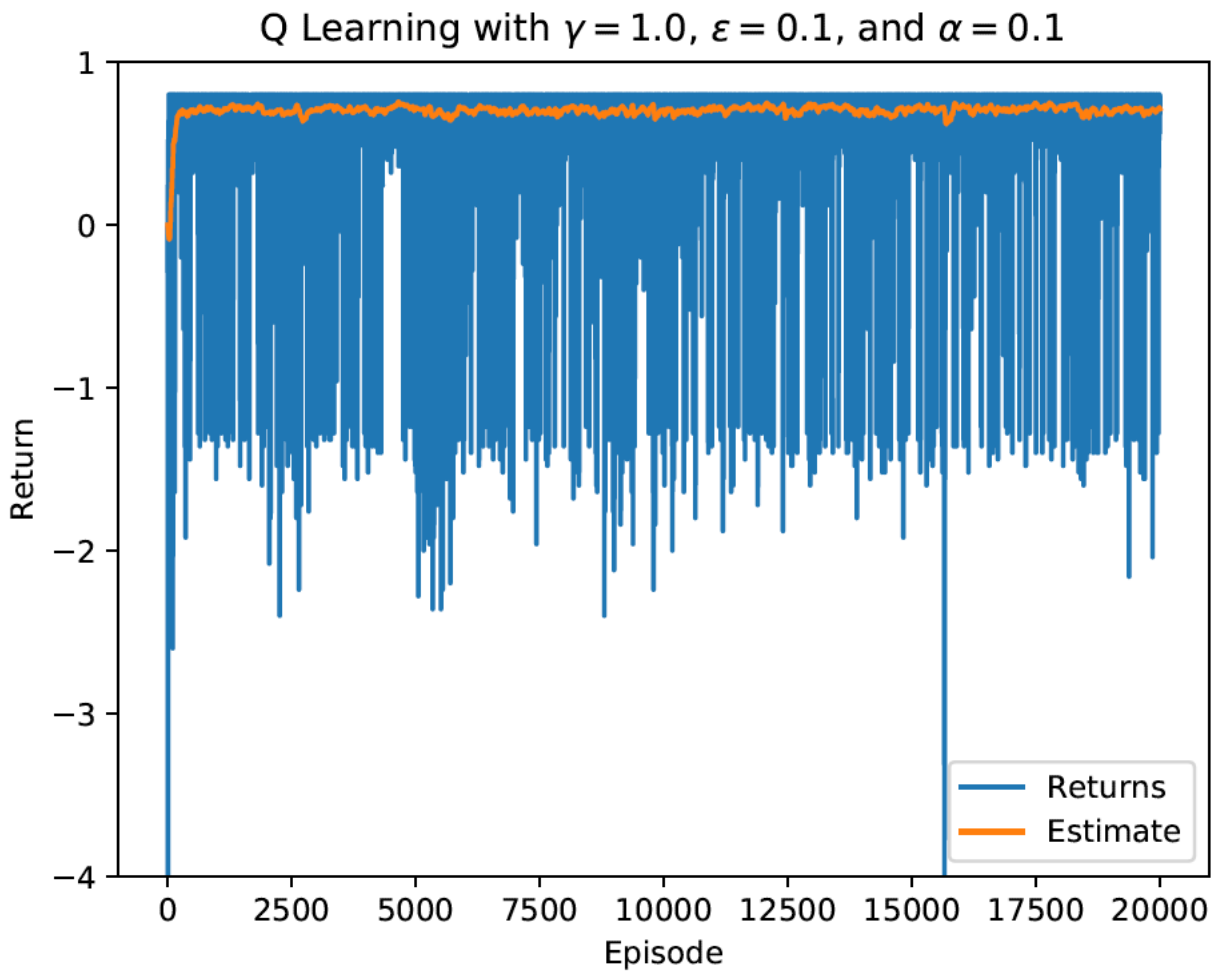
Q_1 and π_1. Show your work. (5 pts)

$$
\begin{aligned}
s &= [2, 0] \\
a &= 0 \\
s' &= [1, 0] \\
\alpha &= 1 \\
\gamma &= 1 \\
R(s) &= -.04 \\
Q_0(s, \cdot) &= [.5, -.1, -.1, .2] \\
Q_0(s', \cdot) &= [-.1, -.1, .4, -.2] \\
\pi_0(s) &= 0 \\
\pi_0(s') &= 2 \\
max_{a'} Q_0(s', a') &= .4 \\
Q_1(s, 0) &= Q_0(s, 0) + (R(s) + max_{a'} Q_0(s', a') - Q_0(s, 0)) \\
&= .5 + (-.04 + .4 - .5) \\
&= 0.36 \\
Q_1(s, \cdot) &= [.36, -.1, -.1, .2] \\
\pi_1(s) &= 0
\end{aligned}
$$

2. Implement Q-learning and have it return U^* and π^*. Display them both here. (15 pts)

   $\pi^* =$ [3. 3. 3. 0. 0. 0. 0. 0. 2. 2. 2.]
   $U^* =$
   (0, 0) : 0.8167918037769507
   (0, 1) : 0.8670920071408602
   (0, 2) : 0.9456598987412232
   (0, 3) : 1.0
   (1, 0) : 0.7572930000439589
   (1, 2) : 0.6039459643373417
   (1, 3) : -1.0
   (2, 0) : 0.7110158779608616
   (2, 1) : 0.6570582921897548
   (2, 2) : 0.5870038870390273
   (2, 3) : 0.36337913694075197

3. Plot the return (reward-to-go) and estimated return ($\max_a Q_k(start, a)$) from the start state for each episode. The estimate should converge to the $U^*(start)$. Use the following parameters for the plot: $\gamma = 1$, $\alpha = 0.01$, $\epsilon = 0.1$, and the number of episodes should be 2000. (10 pts)



4. We will test your code on a grid world that uses different transition probabilities to make sure your implementation is robust. (5 pts)