CS747 FILA
PA 2

Lalit Saini
180070030

October 2021

Contents

1 Task 1

1.1 Implementation Notes

- Transition values and reward values are encoded as dictionary with (initial state, action, final state) as the key. This will save space when there are many transitions with 0 probability like in task 2.
- For value iteration the initial vector is randomly chosen using `np.random.random(number of states)` with as fixed seed. Same is done for initial policy and value vector in HPI.
- Q value of state action pair is calculated to increase speed of execution in VI and HPI. Using this iteration over all states, actions and final states can be bypassed to just iterating over the keys of transition or reward dictionary.
- In HPI, action from improving actions is chosen randomly using `np.random.choice(improving actions)`.
- To obtain Vs in HPI, I didn't solve linear equations as the implementation was supposed to be for sparse and large number of states. I used appraoch similar to VI, converging Vs vector for a fixed policy. The approach can be found here
- Although it is not necessary, I explicitly made values of terminal states as zero if mdtype 'episodic' is encountered.
- Implementation of LP was trickier in terms of increasing execution speed.
- For LP implementation a map of valid actions and final states of an initial states was maintained so to decrease execution time.
- Default algorithm is set to VI.

2 Task 2

2.1 Encoder Implementation

For complete discussion let us assume player P1 is going to play against policy of player P2, i.e. policy of P2 will roll up in environment and P1 will act as agent. We have to find optimal policy for P1.

- We need only two terminal states, 'WIN' and 'DRAW or LOSE'. This gives number of states S, States of P1+2.
- Number of actions still will be 9.
- End states were given last two numbers S-1 and S-2.

- To determine transitions, we will first chose a state of P1. For the chosen state we can take an action corresponding to '0' only. Now will iterate over all possible actions. To determine final state we will first flip the '0' corresponding to action index to '1' and then look into policy of P2 if the obtained state is available or not. If the state is available then according to policy we will flip '0' corresponding to action of P2 to '2' and set the final state as the state obtained. The probability of the transition will the probability of P2 to take that action. Final state obtained by action of P2 if that is in P1 states then reward will be 0, if the final state lead to draw then also reward is zeros only if the final state is such that P1 wins then the reward is 1. Last case is when action of P1 lead to a state that is not in states of P2 then the game is drawn or P1 lost with probability 1 and reward for both cases is 0.
- For invalid moves I have assigned a large negative reward.
- Mdptype will be 'episodic'.
- Discount factor is kept as 1.

2.2 Planner Use

The mdp file generated from encoder is send directly to planner with default algorithm. Planner will give back value function and policy. Important Note: Unlike normal mdp this mdp will have many invalid moves for a state. If planner deduces that no valid action can lead to win it will set V(s) to 0 for all the actions. The argmax function used to find max value action will always return 0 in such case. This is corrected by the decoder.

2.3 Decoder Implementation

Decoder will read the value function and policy and then using state index to name mapping it will generate probability of transitions for that state. If we get V(s)=0 for some state that mean there is no optimal action from that state in that case we will choose all the valid actions with equal probability (if a valid action then probability = 1/a). Decoder will see the states with V(s)=0 and will calculate valid actions and will assign the probability vector. For states in which we have some positive value of value function we will generate probability vector in which optimal action has probability 1 all others have 0.

2.4 Self check of optimality

To check if optimal policy is obtained, I used the obtained policy and simulated it against the other policy of 1000 different samples with different random seeds using a modified version of attt.py .The results are given below (all values in percentage),

Policy	Player	P1 win	P2 Win	Draw
p2_policy1	p1	100	0	0
p2_policy2	p1	71.28	18.22	10.5
p1_policy1	p2	0	100	0
p1_policy2	p2	0.5	99.3	0.2

3 Task 3

For task3 all the policies from 0 to 19 are generated in data folder. Policy naming format is "task3_p<player number 1/2>_policy_<iteration number>.txt", even iteration numbers are for player 2 and odd for player 1. Function "run" was taken from AtttVerifyOutput.py given with code base.

3.1 Procedure

- Started with arbitrary policy of P2 (generated using random function ensuring sum of probabilities equals 1), this policy served as environment for P1 in which it learns its policy.
- New policy of P1 will serve as environment of P2 in which it will learn its new policy.
- After 6 iteration for each policies were observed to converge for any initial random seed.

3.2 Output

The STDOUT will show the path for generated policies. After that it will print yes/no depending on if there is any difference between consecutive polices for a player. Given below is figure of sample output.

```
root@78a9a64991f0:/host# python task3.py
---------Task 3------------
Policy files being created in current directory
P1 policies             P2 policies
task3_p1_policy_1.txt           task3_p2_policy_0.txt
task3_p1_policy_3.txt           task3_p2_policy_2.txt
task3_p1_policy_5.txt           task3_p2_policy_4.txt
task3_p1_policy_7.txt           task3_p2_policy_6.txt
task3_p1_policy_9.txt           task3_p2_policy_8.txt
task3_p1_policy_11.txt          task3_p2_policy_10.txt
task3_p1_policy_13.txt          task3_p2_policy_12.txt
task3_p1_policy_15.txt          task3_p2_policy_14.txt
task3_p1_policy_17.txt          task3_p2_policy_16.txt
task3_p1_policy_19.txt          task3_p2_policy_18.txt
Checking diff between policies
P1 Pi(3)-Pi(1) Diff             P2 Pi(2)-Pi(0) Diff
YES                             YES
P1 Pi(5)-Pi(3) Diff             P2 Pi(4)-Pi(2) Diff
YES                             YES
P1 Pi(7)-Pi(5) Diff             P2 Pi(6)-Pi(4) Diff
YES                             YES
P1 Pi(9)-Pi(7) Diff             P2 Pi(8)-Pi(6) Diff
NO                              NO
P1 Pi(11)-Pi(9) Diff            P2 Pi(10)-Pi(8) Diff
NO                              NO
P1 Pi(13)-Pi(11) Diff           P2 Pi(12)-Pi(10) Diff
NO                              NO
P1 Pi(15)-Pi(13) Diff           P2 Pi(14)-Pi(12) Diff
NO                              NO
P1 Pi(17)-Pi(15) Diff           P2 Pi(16)-Pi(14) Diff
NO                              NO
P1 Pi(19)-Pi(17) Diff           P2 Pi(18)-Pi(16) Diff
NO                              NO
```

Figure 1: Output of task3

3.3 Question

Is the sequence of policies generated for each player guaranteed to converge? If yes, provide a proof of convergence, and describe the properties of the converged policy. If not, either give a concrete counter-example, or qualitative arguments for why the process could go on for ever. If your answer depends on implementational aspects such as tie-breaking, be sure to specify your assumptions.

Answer:

Yes, the policies are guaranteed to converge. We can see this from generated policies in 'data/' after running task3.py, these policies are not changing after 4-5 iteration for each player. Here I started with a stochastic policy as seed after that deterministic policies are obtained by virtue of decoder.

Claim 1: If one of the player reaches an optimal policy in which it always wins, then other player policy will converge for sure.

Argument: Let say P2 achieves the policy in which it always wins. Agent always has an tendency to increase reward, but since the P2 is always going to win then P1 can now only lose the game. So if at any point it gets a losing policy it will not try to improve it further, because in the transitions where P1

can win policy of P2 will give it probability 0 (P2 will not lose). Therefore, other player's policy will also converges if one player's policy converges. Figure below shows empirical proof of the above claim when simulation was run for 1000 game with converged policies.

```
root@807e8a4d0444:/host# python simul.py -p1 task3_p1_policy_19.txt -p2 task3_p2_policy_18.txt
task3_p2_policy_18.txt
Anti-Tic-Tac-Toe Game
Player 1 --- Player 2
P1 %win :0.0, P2 %win :100.0, Draw % :0.0
root@807e8a4d0444:/host#
```

Figure 2: Game on converged policies

Claim 2: One of the player will start to move towards its optimal policy.
Argument: The player which determining its policy for the mdp will assign high probability to an action which leads to win (or will chose that action as policy for the state say). When other player will be determining its policy it will try to not chose the action which will lead to the state from where the previous player with high probability can go to wining state (actually the player is just maximizing its reward but that particular action will have reward 0). In this way probability of chosing an action will keep increasing or will keep reducing to 0. This will push the player towards an optimal policy.
Both the claims show that eventually both players will reach an equilibrium.