Report Lab 1

190050118 - Sudhansh

Task 1

Here is the list of assumptions for this task. All of the algorithms have been implemented in planner.py itself.

Value Iteration

- I assumed a threshold of 1e-9 for convergence. The norm of the difference between values across iterations must be less than the threshold for convergence.
- I have initialized the Value function with 1's for all states.

Howard's Policy Iteration

- I initialized the policy to all 0's. That is, the initial policy takes action 0 for all states.
- Also, I defined a threshold of 1e-3 for the Q values for each action. The policy has improvable actions if the Q value of an action exceeds the value by the threshold.
- Ties are broken among improvable actions by choosing the lowest index.

Implementation details

- Value Iteration is chosen as the default algorithm.
- Value Iteration has been implemented using the B* operator.
- Howard's Policy Iteration consists of two parts. I found the value function of a policy by solving Bellman's optimality equations. The remaining part was implemented using simple python syntax.
- Linear Programming is implemented using the Pulp module provided in Docker. I used the PULP_CBC_CMD solver to solve the constraints of the Linear equation.
- The MDP is defined as a dictionary with the appropriate fields.

Observations:

- LP takes the highest time and VI takes the least time among the algorithms.

Task 2

The encoding and decoding operations are explained below.

encoder.py

- The states in the MDP will be the same as the input states along with an additional end state to capture the “game end” condition. The terminal states are checked using the function checkTerminal.
  Every state is hashed into a corresponding index with the terminal state being mapped to no. of states + 1.
- The state-to-state transitions are effectively captured using the following logic. Consider a transition s to s’ using action a. Now, to find the reward and probability, we do the following:
  - Perform move a on s with the current player and obtain s”.
  - Check whether s” is terminal. If yes, add a transition with probability 1 and the corresponding reward.
  - Otherwise, perform action a’ on s” to obtain s’ based on the opponent’s policy.
  - Add a corresponding transition based on s’.

decoder.py

- Take the policy given by planner.py. Set the probability of actions of policy to 1 and the remaining to 0. Essentially, the policy given by the planner is deterministic.
- As an additional check, ensure all the actions given by the policy are valid. If invalid, it means that the state does not lead to a win condition. Update the policy for such states to a valid action.

Performance

The performance of the algorithm is tested using win rate of the agent in 1000 games. Here are the results in the next page. The results are taken with np.seed(0), and the results may change with different seeds.

We can verify that the algorithms work because we get a 100% winrate in case of deterministic policies.

Player 1 trained against Player 2 with p2_policy1

Player 1	Player 2	Draw
1000	0	0

Player 1 trained against Player 2 with p2_policy2

Player 1	Player 2	Draw
864	103	33

Player 2 trained against Player 1 with p1_policy1

Player 1	Player 2	Draw
0	1000	0

Player 2 trained against Player 1 with p1_policy2

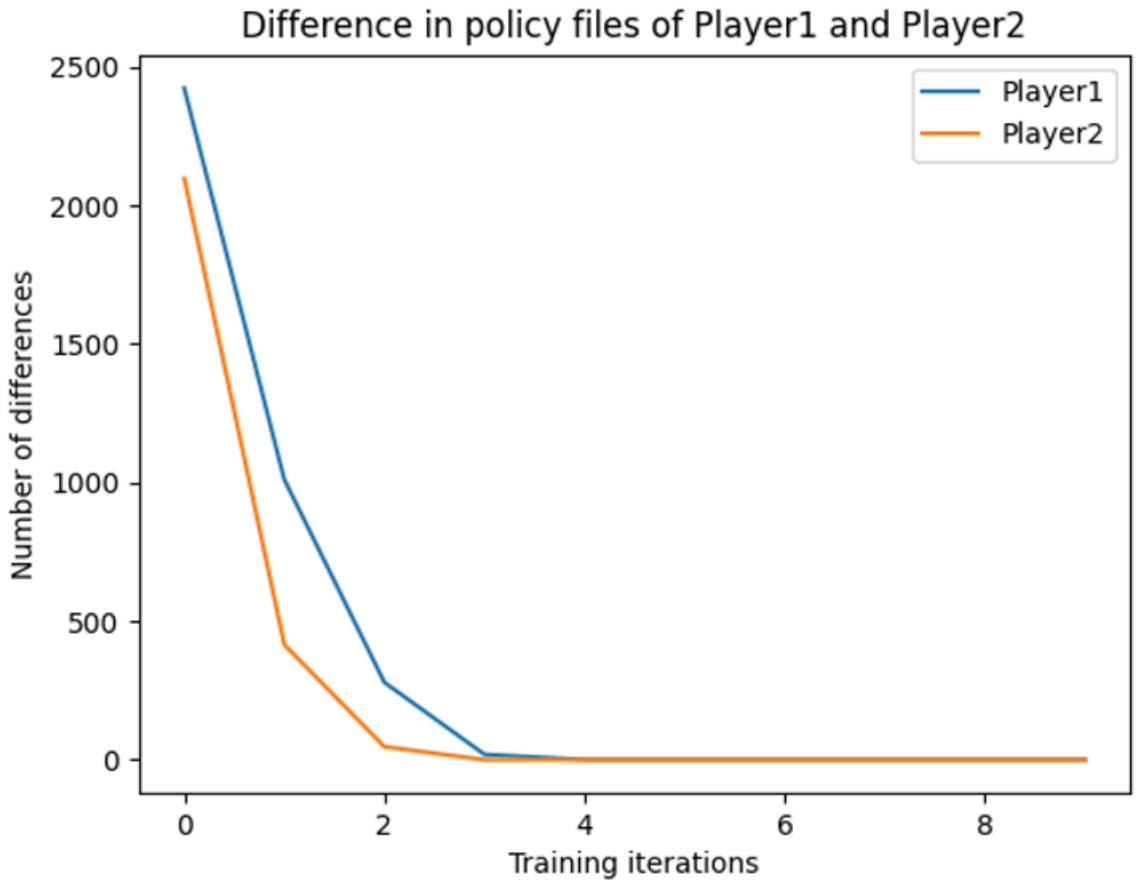
Player 1	Player 2	Draw
1	999	0

Task 3

Implementation

- The loops have been implemented using os.system function in python to call the python scripts.
- Each iteration generates mdp, vp, and policy files. Mdp and vp files are treated as temporary, and are overwritten in each iteration.
- The policy files have names of the form **p?_policy_*** where ? represents the player, and * represents the iteration number.
- 10 files have been generated for each player.
- The training starts with p2's policy initialized to p2_policy2. Then, we train p1 on this policy, p2 on the policy generated by p1, and so on.
- All of the results are stored in **task3** folder.
- Convergence is checked using the difference between files generated across different iterations. The linux function diff is used for the same.

Here is a visualization of the convergence of algorithms, shown as the difference in no. of lines across iterations. The plot is saved as plot.png in the main folder.



It can clearly be seen that convergence is attained by both policies.

Proof of Convergence

Firstly, let us understand why Task2 gives a better policy against the policy we are training on. We can show that the Q values for each state-action pair are nothing but probabilities of winning by taking that action at that state. Since the optimal policy gives the highest Value function, we can say that following the optimal policy gives the highest probability of winning at any given state.
Note. The above mentioned argument can also be proved mathematically.

So, in each iteration of Task3, we obtain the best strategy for the opponent's policy in that iteration. One way of thinking about this is the following. The first policy obtained by our planner in the sequence of policies is deterministic. Suppose the first policy was for player 1. Player 1 decides to place his move in the center of the grid using this policy. Now, when player 2 learns on this policy, it always sees grid with 1 in the center as the start state.
In the next iteration, player 2 learns a policy over this configuration. Because of this deterministic constriction, the state-space seen by the MDP is sort of pruned off. The number of changes keep decreasing, and the policies start converging.

We know that the optimal value function obtained by planner for each MDP is the fixed point of B* operator. Using the definition of B*, we can develop an operator which captures the change in policies for a given player across iterations. More concretely, we can define an operator K which captures the transformation of the policies across iterations in terms of the MDP parameters.

The operator may look something like:

$$(B(F))(s) \;=\; max_{a}\, sum_{s'}\, T(s,\, a,\, s')\,[R(s,\, a,\, s') \;+\; F(s)\,] \text{ where}$$

$$T(s,\, a,\, s'') \;=\; sum_{s'} sum_{a0}\,(T1(s,\, a,\, s')T2(s',\, a0\,,\, s'')$$

Why have we chosen a T of this form? This is the basis on which we generate the MDP using encoder.py. We check all the possibilities to reach a state using both the actions of the current player and the opponent. Therefore, the probability equation looks similar to the one given above.
Now, we can show that this operator is a contraction mapping, and hence has a fixed point. This is quite similar to what we have done in class. We know, due to Banach's fixed point theorem, the repeated application of this operator will lead to convergence. Therefore, convergence is proved.

References have been mentioned in references.txt