Course: Dynamic Programming and Reinforcement Learning **(08-12-2021)**

Submission Assignment 3 - Group #22

Name: Venkat Mohit Sornapudi, *Netid:* 2721697 *Name:* Abhishek Choudhury, *Netid:* 2727712

In this assignment, we address the problem of tic-tac-toe using the MCTS. To find the optimal strategy to play we used UCT and Minimax algorithms. We follow the convention that Player 1 (P1) uses mark 'X' and Player 2 (P2) uses mark 'O'. *To adhere to the page constraint given, we have attached the necessary figures, code snippets, pseudo codes in the Appendix (last section).*

1 UCT - P1 and P2 play optimally and randomly respectively

Tic-tac-toe is played on a 3x3 grid. Here, we represent grid cells and the possible actions as {1,2,3,4,5,6,7,8,9} as shown in Fig 4. A class called Layout was created to build the required environment. Additionally, an interface was created using which we can simulations or play with the trained agent.

MCTS is based on the tree, therefore to implement that, we created a TreeNode whose class and its property is added in the code snippet. The Node has properties that we are using to identify every state during traversal.

Traversal: For traversing the tree, we iterate towards its children using the For-Each loop until we reach the leaf. If the Node is fully expanded, we start propagating back while calculating the score until the ancestor. We address this problem using MCTS with UCT. It has four stages:

1. Selection: We iterate until the node is not terminal in the selection phase. Two cases can be derived: expanded node and unexpanded. If the node is fully expanded, we choose the best move for player two. To calculate the best action, we use the UCB1 formula, which is mentioned in the pseudocode of the appendix. The exploitation factor value is considered as value 2 in this case. We select randomly from all the possible states for the best moves. The other situation is when the tree is not fully expanded; in this case, we explore the state and do the whole expansion.

2. Expansion: For the expansion, we add the nodes for all the possible states. It is done until the tree is fully expanded.To add we iterate to all the possible state until no further states can be added.

3. Rollout: The rollout is a simulation step in which, after identifying the action, we simulation the game for both the players. The optimal player earns a reward of +1 for the win or -1 for the loss. The purpose of performing the rollout is to grant a reward to every node in the trajectory, which is performed in the following step.

4. Backtrack: In the reverse propagation, we update the node score with the score obtained through simulation until its ancestors. This step quantifies the possible actions achieved, further helping in choosing the optimal move.

Obtaining the optimal move: To obtain the optimal move we use the UCB1 formula:

$$\frac{w_i}{n_i} + c \times \sqrt{\frac{\log N_i}{n_i}}$$

1.1 Observations:

- **Winning probability:**If one of the player play optimally then the probability for the following positions 1,3,7,8, and 9 are 66%,51%,66%,40% and 69% on the 5th move. The results are obtained on 1000 simulation for every position.
- **Convergence:** We used the following order for iteration- 100,500, 1000, 5000. We observed that our values starts flattening when the simulation is done for 5000.

2 Minimax method - both P1 and P2 play optimally

For Minimax method, we have imported gym environment and took a few environment helper functions from Kim Jeong Ju's implementation (available on github). Here, the actions are named as {0,1,2,3,4,5,6,7,8} as shown in Fig 5. By default, we reset environment to given initial state (shown in Fig 6).

2.1 Minimax without sampling

Tree construction: To implement Minimax without sampling, we have first had to construct a tree. For this, we used a class called TreeNode (as shown in Fig 9) that has 3 properties: data, children and parent. Here, data contains the current state, next mark (X or O), reward, the action that has brought to this state, level of the node in the tree, and the Q values. The tree is constructed by initializing the root node as the given initial state. From this root node we checked for the existing available moves and saved them as its children. This process is repeated for every subsequent children nodes till we reach all the leaf nodes possible (as shown in Fig 10). In this way we have constructed the full tree that describes all the possible outcomes/states.

Method: In this method, we update the values of the nodes by traversing through the tree in a particular recursive manner (as shown in Fig 11). We start from one of the leaf nodes. Take the reward as the value for the leaf node. Then go its parent node and check if all of its children are updated with the values. If they are not, we traverse down those paths and update the values. If they are all updated, we compare those values and update the value at the current node. If it is a maximizing player turn, we take maximum value of the children values, else take the minimum of them. This process is repeated till we reach the root node.

Results: If both the players play optimally, taking the actions 0, 2, 6, 7, and 8 as the first step (P1- player X) will end in win, draw, win, loss, and draw respectively, with 100% probability. If both players play randomly, taking the actions 0, 2, 6, 7, and 8 as the first step (P1- player X) will result in 66%, 61%, 66%, 33%, and 61% respectively as the win probabilities.

2.2 Minimax with sampling

Here, we sample trajectories from the above built tree for each simulation/iteration. By combining these sampled trajectories we thus obtain new trees. For each such tree we update values in similar way as done in Minimax without sampling. Averaging these values over all iterations (of each corresponding tree) we get the progress of the learned values. To study the convergence, we ran 1000 simulations. The obtained results are shown in Fig 7, 8. We observed that the values are quickly converging in a few hundred simulations. We also observed that if count of sampled trajectories (t) is set less than the total trajectories in the base tree (built in above subsection, say T), the resultant values are less than that of values obtained without sampling, almost by a factor of (t/T). If t is greater than T, we get expected results.

3 Conclusion

- UCT gives good enough approximate values. As we are not traversing the whole tree (with all possible actions and states) in every iteration, it runs faster than Minimax algorithm.
- Minimax gives accurate values than UCT. But, it takes longer to run in comparison.

4 Appendix

```python
class TreeNode():
    def __init__(self, layout: Layout, parent):
        # The matrix for the game - states
        self.layout = layout
        # If the state is a terminal state
        self.isTerminal = True if self.layout.isWin() or self.layout.isMatchDrawn() else False
        # If the tree is fully expanded
        self.isExpanded = self.isTerminal
        # The reference to its parent
        self.parent = parent
        # Total number of visits done
        self.countOfVisits = 0
        # To quantify the score while back propagation
        self.score = 0
        # To have the reference to its children
        self.children = {}
```

Figure 1: TreeNode class

```python
    def select(self, treeNode: TreeNode):
        # Until the node is expanded full
        while not treeNode.isTerminal:
            if treeNode.isExpanded:
                # Exploitation coefficient is 2 for the expanded node
                treeNode = self.optimalMove(treeNode, 2)
            else:
                return self.expand(treeNode)
        return treeNode
```

Figure 2: Select algorithm MCTS UCT

```
    def simulation(self, layout: Layout):
        while not layout.isWin():
            try:
                # Picking randomly
                layout = random.choice(layout.createStates())
            except:
                return 0
        return 1 if layout.playerTwo == Solver.PLAYER_X else -1

    def reversePropagation(self, treeNode: TreeNode, score):
        while treeNode is not None:
            # Update the number of visits
            treeNode.countOfVisits += 1
            # Update the score
            treeNode.score += score
            # Propagate back to its parent
            treeNode = treeNode.parent

    def optimalMove(self, treeNode: TreeNode, ec):
        optimalScore = float("-inf")
        optimalMoves = []
        output: TreeNode
        if len(treeNode.children) > 0:
            for child in treeNode.children.values():
                player = 1 if child.layout.playerTwo == Solver.PLAYER_X else -1
                # Formulae to compute UCB
                actionScore = player * child.score / child.countOfVisits + ec * math.sqrt(
                    math.log(treeNode.countOfVisits / child.countOfVisits))
                # Consider only when it is better than the previous
                # We are picking randomly among the best but not the maximum
                if actionScore > optimalScore:
                    optimalScore = actionScore
                    optimalMoves = [child]
                elif actionScore == optimalScore:
                    optimalMoves.append(child)
            output = random.choice(optimalMoves)
            return output
        return output
```

Figure 3: MCTS UCT

1	2	3
4	5	6
7	8	9

Figure 4: Grid used in UCT

0	1	2
3	4	5
6	7	8

Figure 5: Grid used in Minimax

	O	
X	X	O

Figure 6: Grid used in UCT

Figure 7: Minimax with sampling - Q values

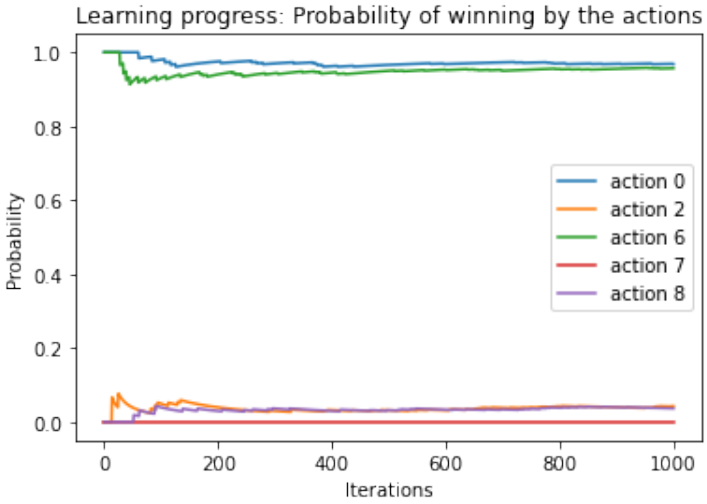

Figure 8: Minimax with sampling - winning probabilities for actions at given initial state

```python
import weakref
class TreeNode(object):
    _instances = set()
    def __init__(self, data, parent=None):
        self.data = data
        self.children = []
        self.parent = parent
        self._instances.add(weakref.ref(self))

    def add_child(self, obj):
        self.children.append(obj)
        obj.parent = self

    def print(self):
        env1 = TicTacToeEnv()
        env1.set_start_mark('X') # start_mark
        intial_state = env1.set_to_state(to_state=self.data['state'])
        env1.render()

    @classmethod
    def getinstances(cls):
        dead = set()
        for ref in cls._instances:
            obj = ref()
            if obj is not None:
                yield obj
            else:
                dead.add(ref)
        cls._instances -= dead

class Tree:
    def __init__(self):
        self.root = TreeNode({'state':"ROOT"})
```

Figure 9: TreeNode

```python
def travarse_tree(current_node, done):

    if not done and current_node.data['ava_actions']:
        action = current_node.data['ava_actions'][0]
        state = env.set_to_state(current_node.data['state'])

        new_state, reward, done, info = env.step(action) # take action
        current_node.data['ava_actions'].remove(action) # removing the action taken from memory

        new_node_data = {}
        new_node_data['state'] = new_state
        new_node_data['level'] = current_node.data['level'] + 1 # increase to new level
        new_node_data['ava_actions'] = env.available_actions() # available actions in new state
        new_node_data['reward'] = reward  # reward obtained in reaching the new state
        new_node_data['action'] = action  # action that has brought to new state
        new_node_data['mms_values'] = {}
        new_node_data['pr_values'] = {}
        new_node_data['mms_progress'] = {}
        new_node_data['pr_progress'] = {}
        if done:
            new_node_data['mm_value'] = reward
        else:
            new_node_data['mm_value'] = None
        new_node = TreeNode(new_node_data)
        current_node.add_child(new_node)  # saving new state values in the new node
        current_node = new_node
        return travarse_tree(current_node, done) # go to next step

    elif current_node.data['action'] != None: # check for reaching root node
        current_node = current_node.parent
        done = False
        return travarse_tree(current_node, done)
```

Figure 10: Building Tree

```python
leaf_nodes = list(get_leaves(root_node))
current_node = list(get_leaves(root_node))[0]

def minimax_without_sampling(current_node):

    if current_node in leaf_nodes:
        parent_node = current_node.parent
        return minimax_without_sampling(parent_node)

    else:  # compare values of children and update current node value
        children = current_node.children
        none_value_children = [child for child in children if (child.data['mm_value']==None)]
        if none_value_children:
            return minimax_without_sampling(none_value_children[0])
        else:
            children_values = [child.data['mm_value'] for child in children]
            if current_node.data['state'][1] == 'X':
                opti_action_idx = np.argmax(children_values)
            else:
                opti_action_idx =  np.argmin(children_values)
            current_node.data['mm_value'] = children_values[opti_action_idx]
            current_node.data['mm_opti_action'] = children[opti_action_idx].data['action']

            if current_node.data['action'] == None:  # check for reaching root node
                print(current_node.data)
            else:
                parent_node = current_node.parent  # after comparision go to parent node
                return minimax_without_sampling(parent_node)
```

Figure 11: Minimax without sampling - Tree traversal