Tic-Tac-Toe Using Q-learning

Vamshi Kumar Kurva

April 20, 2016

Reinforcement Project report

Contents

Chapter 1

The purpose of this document is to provide a report on the implementation of Tic-Tac-Toe using Q-learning and the results obtained.

1.1 Abstarct

In this project we will use reinforcement learning to create a program that learns to play the game tic-tac-toe by playing games against itself. We will consider X to be the maximizer and O to be the minimizer. Therefore, a win for X will result in an external reward of +1, while a win for O will result in an external reward of -1. Any other state of the game will result in an external reward of 0 (including tied games). We will assume that either player may go first.Specifically, we will be using Q-learning, a temporal difference algorithm, to try to learn the optimal playing strategy. Q-learning creates a table that maps states and actions to expected rewards. The goal of temporal difference learning is to make the learner's current prediction for the reward that will be received by taking the action in the current state more closely match the next prediction at the next time step.

1.2 Introduction

1.2.1 Q-Learning

One of the most important breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning.Its simplest form, one-step Q-learning, is defined by

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

the learned action-value function, Q, directly approximates q ,the optimal action-value function, independent of the policy being followed. This dramat-

ically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which stateaction pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated.

1.2.2 Q-Learning Algorithm:

Initialize $Q(s,a), \forall s \in S, a \in A(s)$, arbitrarily, and Q(terminal-state,) = 0
Repeat (for each episode):
Initialize S
Repeat (for each step of episode):
Choose A from S using policy derived from Q (e.g., -greedy)
Take action A, observe R,$S^{'}$
$Q(S,A) \leftarrow Q(S,A) + \alpha[R_{t+1} + \gamma \max_a Q(S^{'},a) - Q(S,A)]$
$S \leftarrow S^{'}$
Until S is terminal

1.3 Implementation Details

Here we are using Q-learning technique to learn the value of(state, action) pairs by using self-play. Here states are the possible board positions of the two players until the game draws or until either of the players wins. Actions are the positions of the grids where one can possibly make a next move. Possible actions in a given state are the empty grid positions.

To learn values of (state,action) pairs in every possible state we are using Q-learning and self-play. Self-play is used to learn the optimum values of (state,action) values before playing against expert human. Tic-Tac-Toe is an episodic task. i.e. each game starts at a random state but will end after finite number of steps.

Since it is the episodic task, rewards will be obtained only at the end of the episode. After every move we will check either the game if finished or the board is full. If the player wins the game, we will reward the winner with +1 and the loser with -1. If the game is a draw, we will reward each player 0.5. While self-play each player will have a table of Q(s,a) pairs because for each player (state, action) values will be different. Each player makes a move which is best according to the current Q(s,a) values. And these values will be updated at the end of the reward based on the reward recieved. After

some finite number of self-play games we would have explored possible states and actions and the Q(s,a) values will be nearer to the optimal values.

Once the self-play is over, when the human makes a move, we will make a move which is best based on the learned Q(s,a) values.

Appendix A

This project is implemented in python.

Code:

```
import random
class TicTacToe:
    def __init__(self, playerX, playerO):
        self.board = [' ']*9
        self.playerX, self.playerO = playerX, playerO
        self.playerX_turn = random.choice([True, False])

    def play_game(self):
        self.playerX.start_game('X')
        self.playerO.start_game('O')
        while True:
            if self.playerX_turn:
                player, char, other_player = self.playerX, 'X', self.playerO
            else:
                player, char, other_player = self.playerO, 'O', self.playerX
            if player.name == "human":
                self.display_board()
            space = player.move(self.board)

            if self.board[space-1] != ' ': # illegal move
                player.reward(-99, self.board)
                break

            self.board[space-1] = char
            if self.player_wins(char):
```

```
                player.reward(1, self.board)
                other_player.reward(-1, self.board)
                break

            if self.board_full(): # tie game
                player.reward(0.5, self.board)
                other_player.reward(0.5, self.board)
                break

            other_player.reward(0, self.board)
            self.playerX_turn = not self.playerX_turn

    def player_wins(self, char):
        for a,b,c in [(0,1,2), (3,4,5), (6,7,8),
                      (0,3,6), (1,4,7), (2,5,8),
                      (0,4,8), (2,4,6)]:
            if char == self.board[a] == self.board[b] == self.board[c]:
                return True
        return False

    def board_full(self):
        return not any([space == ' ' for space in self.board])

    def display_board(self):
        row = " {} | {} | {} "
        hr = "\n-----------\n"
        print (row + hr + row + hr + row).format(*self.board)

class Player(object):
    def __init__(self):
        self.name = "human"

    def start_game(self, char):
        print "\nNew game!"

    def move(self, board):
        return int(raw_input("Your move? "))

    def reward(self, value, board):
        print " rewarded: ".format(self.name, value)
```

```
    def available_moves(self, board):
        return [i+1 for i in range(0,9) if board[i] == ' ']

class QLearningPlayer(Player):
    def __init__(self, epsilon=0.2, alpha=0.3, gamma=0.9):
        self.name = "Qlearner"
        self.harm_humans = False
        self.q = {} # (state, action) keys: Q values
        self.epsilon = epsilon # e-greedy chance of random exploration
        self.alpha = alpha # learning rate
        self.gamma = gamma # discount factor for future rewards

    def start_game(self, char):
        self.last_board = (' ',)*9
        self.last_move = None

    def getQ(self, state, action):
        # To encourage exploration; "optimistic" 1.0 initial values
        if self.q.get((state, action)) is None:
            self.q[(state, action)] = 1.0
        return self.q.get((state, action))

    def move(self, board):
        self.last_board = tuple(board)
        actions = self.available_moves(board)

        if random.random() ¡ self.epsilon: # explore
            self.last_move = random.choice(actions)
            return self.last_move

        qs = [self.getQ(self.last_board, a) for a in actions]
        maxQ = max(qs)

        if qs.count(maxQ) ¿ 1:
            # more than 1 best option; choose among them randomly
            best_options = [i for i in range(len(actions)) if qs[i] == maxQ]
            i = random.choice(best_options)
        else:
            i = qs.index(maxQ)

        self.last_move = actions[i]
```

```
            return actions[i]

        def reward(self, value, board):
            if self.last_move:
                self.learn(self.last_board, self.last_move, value, tuple(board))

        def learn(self, state, action, reward, result_state):
            prev = self.getQ(state, action)
            maxqnew = max([self.getQ(result_state, a) for a in self.available_moves(state)])
            self.q[(state, action)] = prev + self.alpha * ((reward + self.gamma*maxqnew)
                                                           - prev)

if __name__=='__main__':
    p1 = QLearningPlayer()
    p2 = QLearningPlayer()

    for i in range(0,20000):
        t = TicTacToe(p1, p2)
        t.play_game()

    p1 = Player()
    p2.epsilon = 0
    while True:
        t = TicTacToe(p1, p2)
        t.play_game()
```

Bibliography

Richard S. Sutton and Andrew G. Barto,2012. *Reinforcement Learning:An Introduction*, Publisher; The MIT Press,Cambridge, Massachusetts