A study on Tic-Tac-Toe Games

Tony Xiang

October 7, 2019

Contents

1 Introduction and Declarations

Tic-tac-toe game is a very popular and easy game which two players take steps to draw 'O's and 'X's in a 3×3 square. To win this game, a player should draw a line of 'O's or 'X's (diagonal lines are included here). This study is based on Chapter 1 of *Reinforcement Learning: An Introduction, 2nd Edition* and tries to use reinforcement learning as declared in the book. The game is easy to play, but a well-performed model is hard to create.

In the book, each board configuration is called a 'state', and the most important thing is the following formula (state transfering equation):

$$V(s) := V(s) + \alpha(V(s') - V(s))$$

The formula has these meanings:

- The former state is based on the latter state. (Self-memory)
- If a state have more methods to win, the value of it will be closer to 1.

Unlike the book, we let a "tie" outcome have a value of 0.5, a "lose" have a value of 0 and a "win" have a value of 1.

The person goes first will put a 1 in the square, and that goes later will put a -1 in the square. The square is full of 0 when the game starts.

We assume the computer goes first.

2 Implementation

We use Python as programming language.

The board is defined as a 9×1 list, and the winning criterion are defined by slices. The state is defined as a string version of the board list which is identical.

To generate conditions faster, we use *self-playing* method. Using this method might lose some states, but it doesn't matter.

To make the model win most, we use *greedy-playing* method. Using this method will select the highest value in each round. Using this method will lose some states, we work out a way to avoid it.

To train the model multiple times, we use a file to *store all of the states and their values*. We can use a batch to train it multiple times.

2.1 Selecting the playing skills

Two greedy players will quickly reduce the number of states, but will result in losing in real combat (with me). This is because greedy methods will *enhance* a state's value each round. So several routes will stand out from multiple trains, but they can't represent the full states to win. They are *much too greedy*, so something has been neglected.

To amend this, we use a *random-playing* method. Each time, the player randomly select a way, then move onto it, and update the state's value as well. This methods will search for more states as a random selection will likely the search all of the possible ways in a game.

Then we combine the *greedy-playing* and *random-playing* method. Greedy player goes first for a best move, and random player goes later for a broadest move. As greedy move can always find out the best way to deal with every condition, it can increase the probability of winning a game.

And what about *self-playing*? This method also reduces states, but if a greedy player can search for at least **ONE** way to win (or at least tie), it will *enhance* the values alongside this way. Then in real time games, if the greedy player goes first, it will go the best way to win (or at least tie). Moreover, if we put the random player first and the greedy player later, more states will be generated for sure. In this study, we just **take advantage of moving first**.

3 Results and Validations

We add a combat mode in the end to validate this model. The greedy player never loses, and the worst case is a tie.

Here's what we find in the end:

- The greedy player selects the center first.
- If we don't respond in four angles, we will lose. If we do, it will be a tie.

In fact, this is a partial result of this game. In fact, selecting the four angles will end the game just like selecting the center. This validates our model.

A Codes

```
import random
import json

class TTTGame(object):
    def __init__(self):
        self._board = [0] * 9
        self._end = False
        with open('learning.json', 'r') as f:
            self._state = json.loads(f.read())
        self._alpha = 0.05

    def judge(self, state):
        if (sum(state[0: 3]) == 3 or \
            sum(state[3: 6]) == 3 or \
            sum(state[6::]) == 3 or \
            sum(state[0::3]) == 3 or \
            sum(state[1::3]) == 3 or \
            sum(state[2::3]) == 3 or \
            sum(state[0::4]) == 3 or \
            sum(state[2:7:2]) == 3):
            self._end = True
            return 1
        elif (sum(state[0: 3]) == -3 or \
            sum(state[3: 6]) == -3 or \
            sum(state[6::]) == -3 or \
            sum(state[0::3]) == -3 or \
            sum(state[1::3]) == -3 or \
            sum(state[2::3]) == -3 or \
            sum(state[0::4]) == -3 or \
            sum(state[2:7:2]) == -3):
            self._end = True
            return 0
        elif 0 not in state:
            self._end = True
            return 0.5 # can be set to 0 if you need sharper
            ↪  winning criterion.
        else:
            self._end = False
            if str(state) not in self._state:
                self._state[str(state)] = 0.5 # move state
            return self._state[str(state)] # study starts from
            ↪  here ...

    def random_move(self, move_type=-1):
```

```
        self.judge(self._board)
        if (self._end):
            return '[End]'
        empty = []
        count = 0
        for val in self._board:
            if (val == 0):
                empty.append(count)
            count += 1
        select = empty[random.randint(0, len(empty) - 1)]
        move_board = self._board.copy()
        move_board[select] = move_type
        value = self.judge(move_board)
        self._state[str(self._board)] =
        ↪  self._state[str(self._board)] + self._alpha * (value
        ↪  - self._state[str(self._board)]) # update move
        self._board = move_board.copy()
        return select

    def greedy_move(self, move_type=1):
        self.judge(self._board)
        if (self._end):
            return '[End]'
        selects = []
        max_value = -1
        count = 0
        for val in self._board:
            if (val == 0):
                move_board = self._board.copy()
                move_board[count] = move_type
                value = self.judge(move_board)
                if (value > max_value):
                    selects = [count]
                    max_value = value
                elif (value == max_value):
                    selects.append(count)
            count += 1
        select = random.sample(selects, 1)[0]
        move_board = self._board.copy()
        move_board[select] = move_type
        value = self.judge(move_board)
        self._state[str(self._board)] =
        ↪  self._state[str(self._board)] + self._alpha * (value
        ↪  - self._state[str(self._board)]) # update move
        self._board = move_board.copy()
        return select
```

```
    def play(self):
        self._board = [0] * 9
        self._end = False
        while not self._end:
            s1 = self.greedy_move()
            s2 = self.random_move()
            # print('greedy selection:', s1, 'random
            ↪  selection:', s2)

    def train(self, epoch=1000):
        for i in range(0, epoch):
            self.play()

    def dump_state(self):
        with open('learning.json', 'w') as f:
            f.write(json.dumps(self._state))

    def pretty_print_board(self):
        print(self._board[0], self._board[1], self._board[2])
        print(self._board[3], self._board[4], self._board[5])
        print(self._board[6], self._board[7], self._board[8])

    def combat(self):
        self._board = [0] * 9
        self._end = False
        while not self._end:
            s1 = self.greedy_move()
            self.pretty_print_board()
            print("Winning prob:", self.judge(self._board))
            if (self._end):
                print('You lose / a tie!')
                break
            s2 = input('Please enter your move: ')
            while self._board[int(s2)] != 0:
                s2 = input('Please enter your move: ')
            self._board[int(s2)] = -1
            self.pretty_print_board()
            print("Winning prob:", self.judge(self._board))
            self.judge(self._board)
            if (self._end):
                print('You win!')


if __name__ == '__main__':
    tttg = TTTGame()
```

```
tttg.combat()
tttg.train(100000)
tttg.dump_state()
```