Machine Learning Workshop
Tic-Tac-Toe Project

Martin Mrugała
Patryk Walczak
Filip Szymczak
Bartek Żyła
Maciej Zalewski

June 16, 2020

Contents

Introduction

1.1 Standard Tic-tac-toe

According to the definition in the Oxford Dictionary of English, Tic-tac-toe is a game in which two players seek to complete a row of either three noughts or three crosses drawn alternately in the spaces of a grid of nine squares.

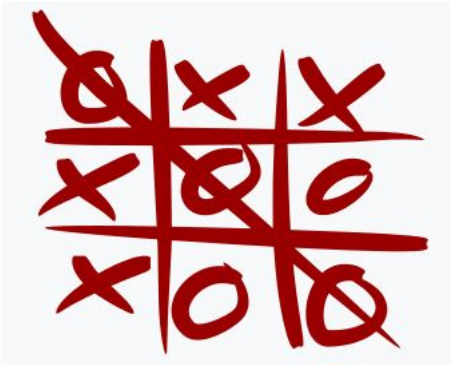

Figure 1.1: A completed game of Tic-tac-toe[1].

And as it turns out, there are 255 168 possible games. So this means that, with the help of the reinforcement learning, a bot may train to mastery.

[1]Source of the image: `https://en.wikipedia.org/wiki/Tic-tac-toe`

1.2 Our implementation

Our team decided to create a bot which would play this game but in an expanded version of it. The principal changes concern, inter alia:

- **The size of the grid could be infinite**
  The number of squares is not boundless, of course. Due to the CPU, as well as the storage limitations, the infinity has to be simulated. We came up with two solutions. The first one is pretty straightforward, but a little preachy. The idea is to set the size in advance. The second one, in turn, assumes an increase in the size whenever the players get close to the edge.

- **Condition of winning**
  The length of the sequence of X's or O's needed to win may have any value. Naturally, it has to satisfy the following condition:

  $$0 < length < edgelength$$

  There is still the matter of the draw. In case of the constant size gird, the tie occurs, whenever there is no way for any player to win.

1.3 Machine Learning backgroud

To deal with the problem we are using reinforcement learning, which trains algorithm by a system of reward and punishment. In this way we can put one bot against the other and let them play, so they will find the best solutions to the game by themselves without any supervision.
Basic reinforcement learning can be modeled as a Markov decision process:

- set of states S (state space)
- set of actions A (action space)
- reward R after transition from state s to s' under action a
- probability of transition from state s to s' under action a

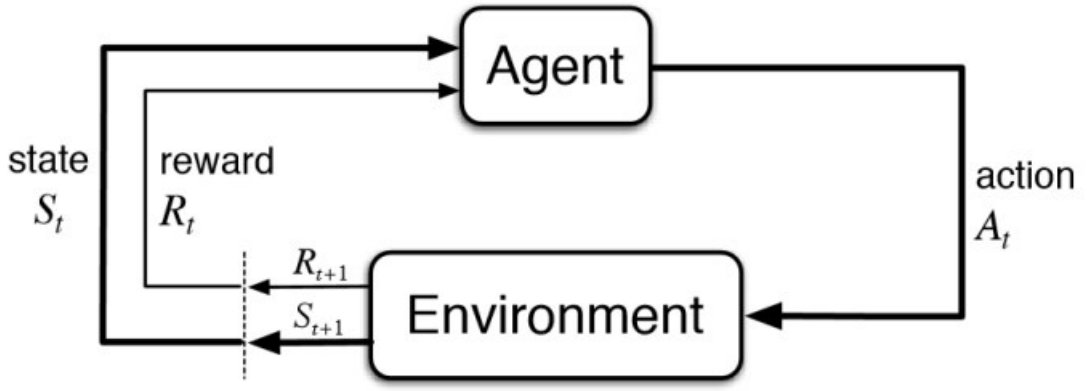

Figure 1.2: Decision process diagram[2].

A reward is a scalar feedback signal, which indicates how well agent is doing at the given step R_t. Reinforcement is based on the reward hypothesis, which tells that goals can be described by the maximisation of expected cumulative reward, so the agent's job is to maximise cumulative reward. To do so, in our implementation, it selects actions, which gives the greatest reward (have the greatest value) immediately.

Observation is the information about environment received by the agent. The history is the sequence of observations, actions and rewards up to time t.

$$H_t = A_1, O_1, R_1, ..., A_t, O_t, R_t \tag{1.1}$$

State is the information used to determine the next action. Our environment (game board) is fully observable, so the state of environment is the same as the state of the agent and observation of the agent. State can be described as a function of history.

$$S_t = f(H_t) \tag{1.2}$$

A state S_t is Markov state if and only if

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, ..., S_t] \tag{1.3}$$

Then if the current state is known, the history is not needed. State S_t is enough to decide on the next action. Additionaly, our environment is fully

[2]Source of the image: `https://www.kdnuggets.com/images/reinforcement-learning-fig1-700.jpg`

observable. It means that the agent state is equal to the environment state, the agent has all information about environment.

Reinforcement learning agent contains two components: policy and value function. The policy describes how the agent behaves, it is a map from state to action. Value function tells how good is each state or action.

In our implementation the policy is a dictionary containing every state encountered during learning with some value. Value function decides which move is the best by analysing these values for the next actions, but only one ahead. Our agent is using a deterministic policy, the action is chosen by finding the greatest value for the next state. There is only a slight probability for doing a random action, thanks to this the algorithm can learn and explore new paths.

Solution

We have prepared two basic versions, for finite and infinite boards, and one which is doing smarter moves for the infinite board.

2.1 Finite board

Bot algorithm is trained against the other instance of the bot at the beginning, then the winner is chosen by comparing the number of wins and forwarded to the player as the best current solution on the proper position (O or X). Iterations are the number of games to play between bots - the more they play, the better they are.

```
def train(iterations):
        player1Win = 0.0
        player2Win = 0.0
        loadPolicy()
        for i in range(0, iterations):
                play()
                if (player1 wins):
                        player1Win += 1
                        player1.feedReward(1)
                        player2.feedReward(0)
                if (player2 wins):
                        player2Win += 1
                        player1.feedReward(0)
                        player2.feedReward(1)
```

```
                if (draw):
                        player1Win += 1
                        player1.feedReward(0.1)
                        player2.feedReward(0.1)
        savePolicy()
        if (player1Win > player2Win):
                return player1
        else:
                return player2
```

Limiter in the bot instance limits the range of the optional moves on the board. ExploreRate is a probability of a random action, stepSize controlls increasement of the value for the given state during reward feeding.

In this function, the set of actions is constructed for a given state. AI decides which move to make by considering every possibility in it's current bounds. The move is analyzed by checking how the state would look like after the action. If such situation on the board has not occured before (it is not in the estimations), the new state is given the value (1 for win, 0 for lose and 0.5 otherwise), hashed and added to the estimations dictionary.

```
def move(state):
    nextStates[ ]
    nextPositions[ ]
      for i in range (-limiter, limiter+1):
        for j in range (-limiter, limiter+1):
          if move is possible:
            nextPositions.append((i,j))
            nextStates.append(nextState(i,j).getHash())
            if nextState(i,j).getHash() not in estimations:
            if nextState(i,j).isEnd:
              if win:
                estimations[nextState(i,j).getHash()] = 1
              else:
                estimations[nextState(i,j).getHash()] = 0
            else:
              estimations[nextState(i,j).getHash()] = 0.5
```

After calculations the decision has to be made. There is a chance to perform a random action (probability of this is given by exploreRate), which allows to explore new paths with a possibility of finding a new, better tactic. Otherwise the action with the biggest value based on the previous training is chosen.

```
if random.binomial(1, exploreRate):
        r = random.(0, length(nextPositions)-1)
        action = nextPositions[r]
        states.append(nextStates[r])
        return action

values = [ ]
for hash, position in (nextStates, nextPositions):
        values.append((estimations[hash], position))
v = value.index(max(values.estimation))
action = values[v][1]
states.append(nextStates[v])
return action
```

Upon finishing the game iteration, algorithm is fed with the reward according to the result. Values for states from the path chosen in the current iteration are adjusted starting from the last one. The change depends on the "size" of the reward, stepSize and values of previously rewarded states along the path when going back. That is how the algorithm learns which paths are better to choose and which ones should be rather ommited. If the outcome was positive, values will be slightly increased. If the game was lost, values will drop down a little bit. The changes are scaled so they are small and do not affect strongly the possibility of choosing other paths. Parameters can be changed to find the optimal way of learning.

```
def feedReward(reward):
  if length(states) == 0:
    return
  target = reward
  for latestState in reversed(states):
    value = estimations[latestState]
                + stepSize * (target - estimations[latestState])
    estimations[latestState] = value
    target = value
```

There are two policies, optimal policy 1 is for the starting bot and optimal policy 2 is for the bot which has second move. Each training session starts with loading the last optimal policy as estimations and ends with saving them.

The file contains hashed states, which allows for a quick comparison as keys, and their appropriate values in a dictionary. In this way we do not

have to store and compare the whole state, calculating hash is enough to preserve the uniqueness of the state. Only the set of moves is being hashed, the order is not important. Stored data should be reduced to the necessary minimum because the number of possible states grow very quickly (for the infinite board there is an infinite number of states).

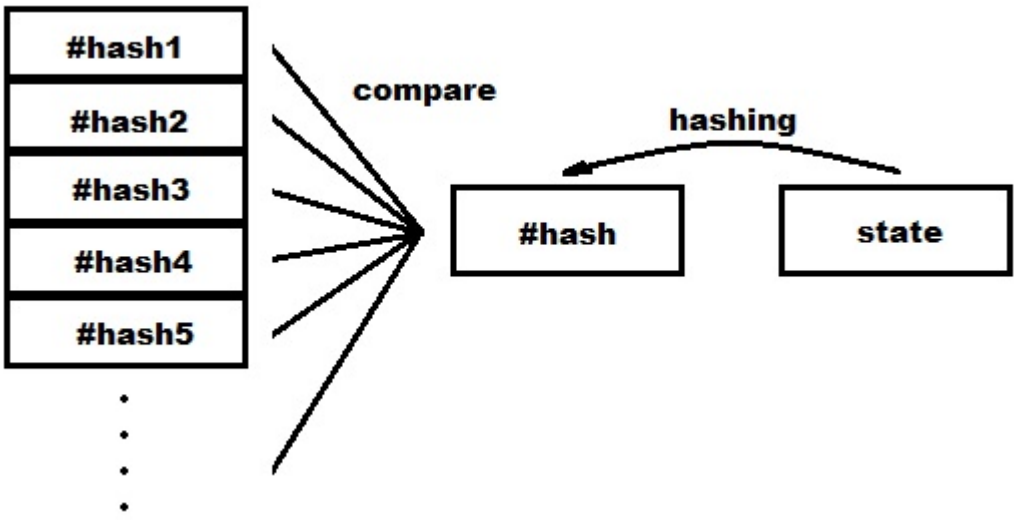

Figure 2.1: Comparision of a given state with the estimations.

2.2 Infinite board

For the infinite board the agent is the same as for the finite one. The only change is in the *move*() function. To deal with the infinity, if there are no more possible positions in the currently considered bounded area, boundaries are extended and the process continues. This is done before choosing the next move. Bounds value is equal to the limiter starting value.

```
def move(state):
        ...

        if length(nextPositions) == 0:
                limiter += bounds
                return move(state)

        ...
```

2.3 Infinite board (smarter moves)

On the infinite board there are infinite possibilities for the next move. To help a little bit our agent in playing and learning process, we are limiting the set of possible actions by eliminating ones which are pointless from the logical point of view.

To be exact, the set of possible moves consists of possitions which are in the neighborhood (described by *limiter* value here) of already taken positions. In that way we will not have situations where the bot randomly decides to make a move on some position far away from the center of the action, which is ineffective and illogical.

Additionaly, if the bot is starting the game, the first move will be at a random position in the middle of the board (3x3 square).

```
def move(state):
  nextStates[ ]
  nextPositions[ ]

  if length(state.data) == 0:
    i = random.randint(0,3)
    j = random.randint(0,3)
    nextPositions.append((i,j))
    nextStates.append(state.nextState(i,j).getHash())

  ...

  else:
    for position in state.data:
      for i in range (-limiter, limiter+1):
        for j in range (-limiter, limiter+1):
          ii = position[0] + i
          jj = position[1] + j
          if (ii,jj) not in state.data:
             nextPositions.append((ii,jj))
             nextStates.append(state.nextState(ii,jj).getHash())

  ...
```

Results

3.1 Predictions

As it was mentioned at the beginning, the number of possible games on the 3x3 board is 255 168. When the size increases, the number of possible games skyrockets to the unimaginably large number. It is not that hard for the algorithm to find all the possible states of the game for the fairly small boards, but every small change to the board size generates a lot of new possibilities. The minimal number for moves to win, if the winning condition is 3 in a row, is 5 and maximum is the number of positions on the board. For each number of moves we have to consider every possible state of the game. For the infinite board this number goes to infinity.

Win in 5 moves	**1440**	0.6%
Win in 6 moves	**5328**	2.1%
Win in 7 moves	**47952**	18.8%
Win in 8 moves	**72576**	28.4%
Win in 9 moves	**81792**	32.1%
Draw	**46080**	18.1%
Total	**255168**	**100.0%**

Win in 5 moves	**172**
Win in 6 moves	**579**
Win in 7 moves	**5115**
Win in 8 moves	**7426**
Win in 9 moves	**8670**
Draw	**4868**
Total	**26830**

Figure 3.1: Combinations of moves for a 3x3 board[1]

Theoretically our bot should be able to achive skill level which will be enough to play on a par with a human. Unfortunately, as we expected, to do so, even on a finite board, the agent should go through more iterations than we were capable of running. Policy file quickly grows to the huge size and we do not have enough computational power to train it well enough.

While it is not capable of playing with a human, results of our training show that the algorithm is working correctly and we can assume that after enough iterations it should play as expected.

This statement is based on results of two bots playing against each other and fact that the player who starts has better chances of winning in Tic-Tac-Toe, because the second one has to react to his moves. If the first one does not make any mistake in the worst scenario, on the finite board, the result of the game will be a draw. For the infinite board there are no draws, so the starting player should always win.

3.2 Finite board

Parameters and results for the tests:

iterations = 10 000 000

boardSize =

need for win =

limiter =

stepSize =

exploreRate =

Player1 wins =

Player2 wins =

Obtained results were satisfying and as expected. Both bots (trained for Player1 and Player2) were losing to a human player but their actions were not completely random. At the end of the training the number of games which were won by the bot with first move was approximately 2/3 of all played games. On the other hand after only a few thousands it was slightly more than 1/2. It shows that the algorithm works for the finite board, because the

[1]Source of the image: `http://www.se16.info/hgb/tictactoe.htm`

starting player have an advantage and, without any mistakes, should always win or at least have a draw. With more iterations we can expect to achieve better results, but the improvement will be slower than at the beginning.

3.3 Infinite board

Parameters and results for the tests:

iterations = 10 000 000

boardSize =

need for win =

limiter =

stepSize =

exploreRate =

Player1 wins =

Player2 wins =

For this version results were not satisfying. Bots were losing to a human player and their moves didn't make too much sense. Clearly expanding the approach from the finite version only by increasing the area for possible moves was not enough. Still, the starting bot (Player1) was better than the second one.

3.4 Infinite board (smarter moves)

Parameters and results for the tests:

iterations = 10 000 000

boardSize =

need for win =

limiter =

stepSize =

exploreRate =

Player1 wins =

Player2 wins =

After changes which limit the actions to the "center" of the game, results are much better. They are able to choose next moves more wisely than before. It is also easier for them to deal with the infinite possibilities of moves.

3.5 Summary

Our implementation of reinforcement learinng to build a bot for the Tic-Tac-Toe game works quite well when it comes to play with the other bot and meets our expectations. When playing together, the first bot is winning significantly more often.

On the other hand playing with a human is not recommended at this moment, because bots still need more training and tests. Unfortunately, we are not able to perform such computations. To deal with this problem we could try to find more powerful computer or try to optimalise the bot further.

Creating a bot which will be able to make decisions on the infinite board requires a lot of training iterations. Learning to play on a finite board is much faster. From possible changes, we could add choosing the best move randomly if there is more than one with the same maximum value.

Instruction

There are three versions of the program, each in their respective order. Player makes move by giving two integers, respectively x and y position.

To start the training run the *game.py* python script, at the end the results will be shown and the game will start with the better bot.

To play without the training run *only_play.py* python script. Important note: To do so, policy from the previous training is needed. Otherwise the bot will not know how to play at all. Player can be changed manually in the code of this script.

Bibliography

Reinforcement Learning Lectures by David Silver on YouTube

https://en.wikipedia.org/wiki/Reinforcement_learning

https://github.com/JaeDukSeo/reinforcement-learning-an-introduction/tree/master/chapter01