Machine Learning Workshop
Tic-Tac-Toe Project

Martin Mrugała
Patryk Walczak
Filip Szymczak
Bartek Żyła
Maciej Zalewski

June 14, 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.

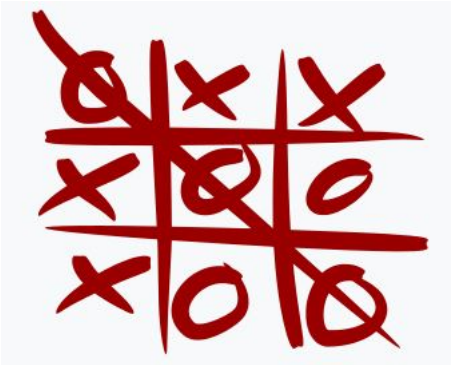

Figure 1.1: A completed game of Tic-tac-toe[1].

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 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

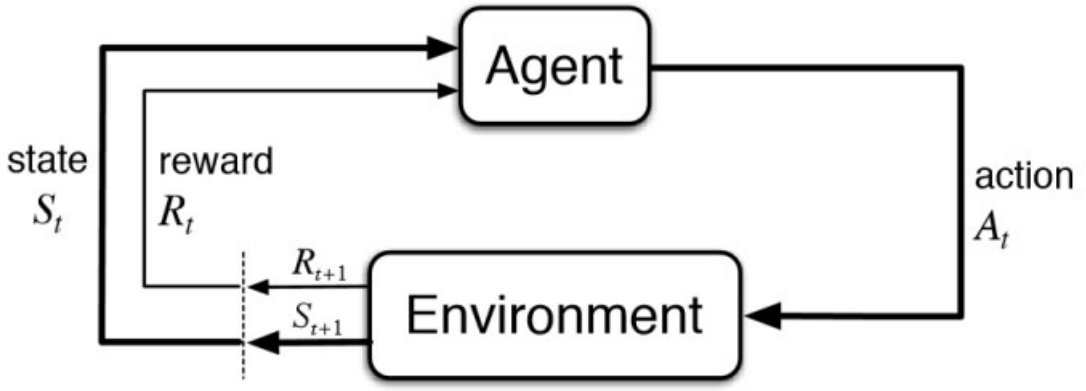

Figure 1.2: Decision process diagram[2].

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. It can be described as a function oh history.

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

A state S_t is Markov state if and only if

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

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

Solution

2.1 Algorithms and data processing functions

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. 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 list of moves, in proper order for the given state, is being hashed. Stored data should be reduced to the necessary

minimum because the number of possible states grow very quickly (for the infinite board there are infinite states).

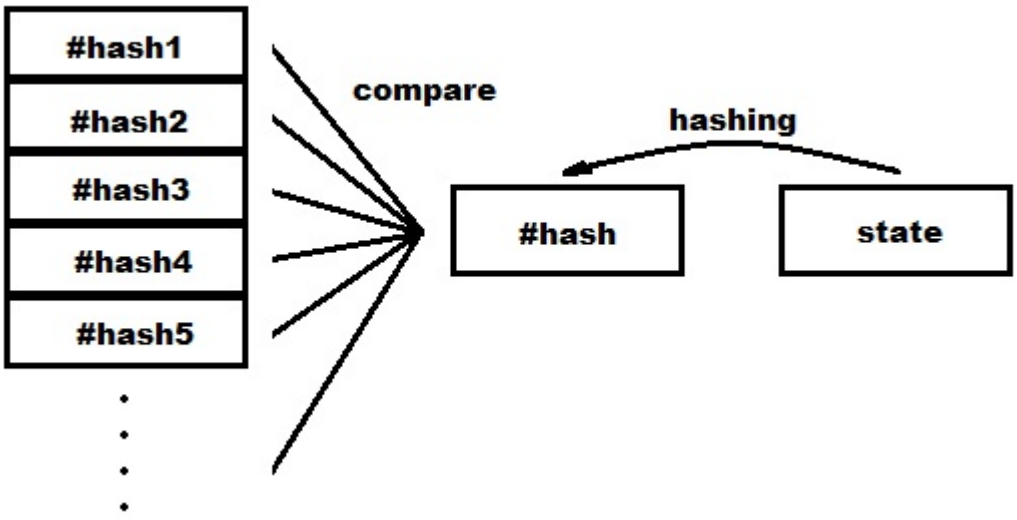

Figure 2.1: Comparision of a given state with the estimations.

Code

```
class MyClass(Yourclass):
def __init__(self, my, yours):
bla = '5 1 2 3 4'
print bla
```

Bibliography