Machine Learning Engineer Nanodegree

Capstone Project

Implementation of Tic-Tac-Toe Agents using Reinforcement Learning techniques

Demetrios Vassiliades

February 25th, 2017

I. Definition

Project Overview

The goal of this project is to make the machine learn how to play Tic-Tac-Toe, in a board of varying sizes and number of winning seats (winning line size).

The project's domain background is Reinforcement Learning, which is an area of Machine Learning. Reinforcement Learning techniques aim at enabling machines to determine the ideal behavior within an environment, in order to maximize some notion of cumulative reward.

The strategy to achieve the project's goal was to initially implement the game engine component, a fully functional Tic-Tac-Toe game, and then to implement simple player components, such as the Human Interaction component and the Random Move Selection component. The last and most important part of the project was the implementation of the player components with Reinforcement Learning algorithms.

The algorithms created in this project are mostly based on Q-Learning techniques, which can be used to determine an optimal action-selection policy for a given Markov decision process.

Related research on this domain has been made by artificial intelligence companies, such as DeepMind with Atari Games [1] and AlphaGo [2] [3] projects, where Reinforcement Learning was combined with Deep Neural Networks.

Problem Statement

The widely known Tic-Tac-Toe game is a solved game [4] played by two players on a 3x3 board. Each player marks a seat on the board with his symbol. The purpose of the game is to fill a line of consecutive seats marked by the same symbol. Each player tries to form either a horizontal, vertical or diagonal line with his symbol. At the same time the player tries to prevent the opponent from forming a line first. The latter is also known as Minimax [5]. The game can be extended to boards with bigger number of rows and columns, with the same rules.

The techniques used in this project can be applied on numerous real-world applications, where there's a need for a machine to achieve a certain goal (e.g. autonomous driving to safely reach a certain destination) and to minimize the possible loss for a worst case scenario (e.g. minimizing casualties upon a collision).

Metrics

The metrics of this project are collected after a number of games among the different algorithms developed. The outcome of these games, the time used for training and the memory footprint (maximum memory usage of the process) for each variation are the metrics that will form the benchmark model of this project. Among algorithms that outperform the benchmark, the most efficient algorithm is the one that requires the less resources (CPU, memory), as indicated by the Occam's razor [6].

1 Mnih, Volodymyr et al. "Playing Atari with Deep Reinforcement Learning" arXiv preprint arXiv:1312.5602 [cs.LG]
2 "AlphaGo" https://deepmind.com/research/alphago/
3 Silver, David et al. "Mastering the game of Go with deep neural networks and tree search" Nature 529, 484–489 (28 January 2016)
4 "Solved Game" https://en.wikipedia.org/wiki/Solved_game
5 "Minimax" https://en.wikipedia.org/wiki/Minimax
6 "Occam's Razor" https://en.wikipedia.org/wiki/Occam's_razor

II. Analysis

Data Exploration and Visualization

The Tic-Tac-Toe environment consists of one game engine and two players.

In the beginning of each game, the game engine is instantiated with a number of parameters. The parameters are:

- Board (edge) size, at least three (3)
- Winning seats (Winning line size), the number of consecutive seats that a player has to mark in order to win. The winning line size cannot be larger than the number of rows or columns of the board. The winning line can be any horizontal, vertical or diagonal sequence of seats held by a player.
- Player 1, an instance of an object type player
- Player 2, an instance of an object type player

The board is constructed as a two-dimensional list, and its size is defined by the number of rows and columns. An example of a 3x3 board is depicted in Figure 1.

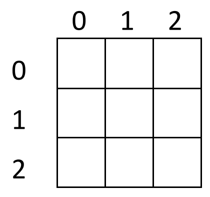


Figure 1: Initial state of a 3x3 board

Each seat will have a default value of *None*, which signifies that the seat has not been marked by any player yet. The game engine is responsible to maintain the status of the board, update it after each player's input, which will be the selected seat notated as (row, column), and decide the result of the move. The result of each move can be one of the following:

- If a player wins, then both players will be informed about the winner (Figures Figure 2 and Figure 3)
- A draw (Figure 4)
- The game is still ongoing (Figure 5)

	0	1	2
0	X	X	X
1		O	
2	O		

Figure 2: Player 1 (X) wins

	0	1	2	3
0		X	O	
1		O		X
2	O		X	
3				

Figure 3: Player 2 (0) wins in a 4x4 board and a winning line size of 3

	0	1	2
0	X	O	X
1	X	O	X
2	O	X	O

Figure 4: Draw

	0	1	2
0		X	
1	O		
2			

Figure 5: Ongoing game

The game engine asks from the next player to make a move, by providing the player with the current state of the board. Each player will have to decide the next move and store the state of the board, as well as the next move made.

When the game reaches a final state (draw or win), the game engine will inform both players about the result of the game. Each player will map the final result of the game to a numeric value that will represent the reward or the penalty.

Algorithms and Techniques

The solution to the problem stated in this project is an algorithm that enables the machine to find out the available moves and decide the next move by applying Reinforcement Learning techniques. The algorithm is implemented on the player's component.

There are three variations of the solution algorithm presented. All algorithms use Monte Carlo method but the difference between them is the way they update and store the learning values.

In general, Monte Carlo methods can be used in an algorithm that mimics policy iteration, which consists of two steps: policy evaluation and policy improvement. The Monte Carlo method is used for the policy evaluation step. In all variations there is a variable ε (epsilon), which represents the possibility of choosing a random move, rather than choosing the move with the largest Q-value. For each random move, the ε variable will be decreased. The value deducted from ε is calculated by the following:

$$\varepsilon \leftarrow \varepsilon * (1 - step),$$

where *step* is a value between (0,1).

The first algorithm, which is a naïve implementation of the Q-learning technique, updates the learning values in the end of the game, based on the outcome and by applying the reward retrospectively on each move that the player made during the game:

$$Q(s,a) \leftarrow \frac{Q(s,a) * t + r}{t+1},$$

where:

s: state of the board

a: action

t: the number of times passed from the set of (s,a)

r: final reward

The final reward of the last action will be calculated based on the result of the game.

The result of each game for all variations of the algorithm will be mapped to a numeric value, based on the following map:

- Win: 1.0
- Draw: 0.5

- Lose: 0.0

The initial value for each set of state and action will be 0.5.

The second algorithm is based on the Q-learning technique and updates the learning values after each move of the player, based on the reward of each move.

$$Q(s,a) \leftarrow Q(s,a) + alpha * (max(Q(s',a')) - Q(s,a)),$$

where:

s: state of the board

a: action

alpha: learning rate (0, 1]

s': next state of the board

a': next move

In the above equation, the reward is represented by the maximum value for the next move in the next state (max(Q(s', a'))).

The third variation uses a neural network and is implemented with Tensorflow, a deep learning framework. The neural network consists of three layers: the input layer, the hidden layer and the output layer (Figure 6). The input layer receives as input the player's state, e.g. the seats marked by the player, the opponent's state, e.g. the seats marked by the opponent, and the last move taken by the player. Each of these inputs has a tensor size equal to the board size. The hidden layer is a 1-layer, which consists of a number of nodes defined by the variable *hidden_layer_size*. The output layer, the size of which is equal to the board size, contains the Q-Values. In order to identify the next best move, the greedy algorithm will look for the seat with the maximum Q-Value. For the training of the neural network, an optimizer is created, using either *GradientDescentOptimizer* or *AdamOptimizer*. The optimizer tries to minimize the loss, which is defined by the following:

```
self.loss = tf.reduce_mean(tf.square(self.learned_value - action_q_values))
```

The method *reduce_mean* is a reduction method that computes the mean of elements across dimensions of a tensor. The input of this method is the square of the result obtained by subtracting the *action_q_values* from the *learned_value*.

The variable *learned_value* is defined by the following:

$$learned\ value \leftarrow gamma * \left(max(Q(s',a'))\right),$$

where:

- *gamma*: the discount factor
- *s'*: next state of the board
- *a'*: next move

In case of a final status of the game, the learned value is the reward or the penalty.

The variable *action_q_values* is defined by the following:

```
action_q_values = tf.reduce_sum(tf.mul(self.q_values_nn, self.move), reduction_indices=[1, 2]
```

The method *reduce_sum* is a reduction method that computes the sum of elements across dimensions of a tensor. The input of this method is the element-wise multiplication of the (reshaped) output layer with the selected move.

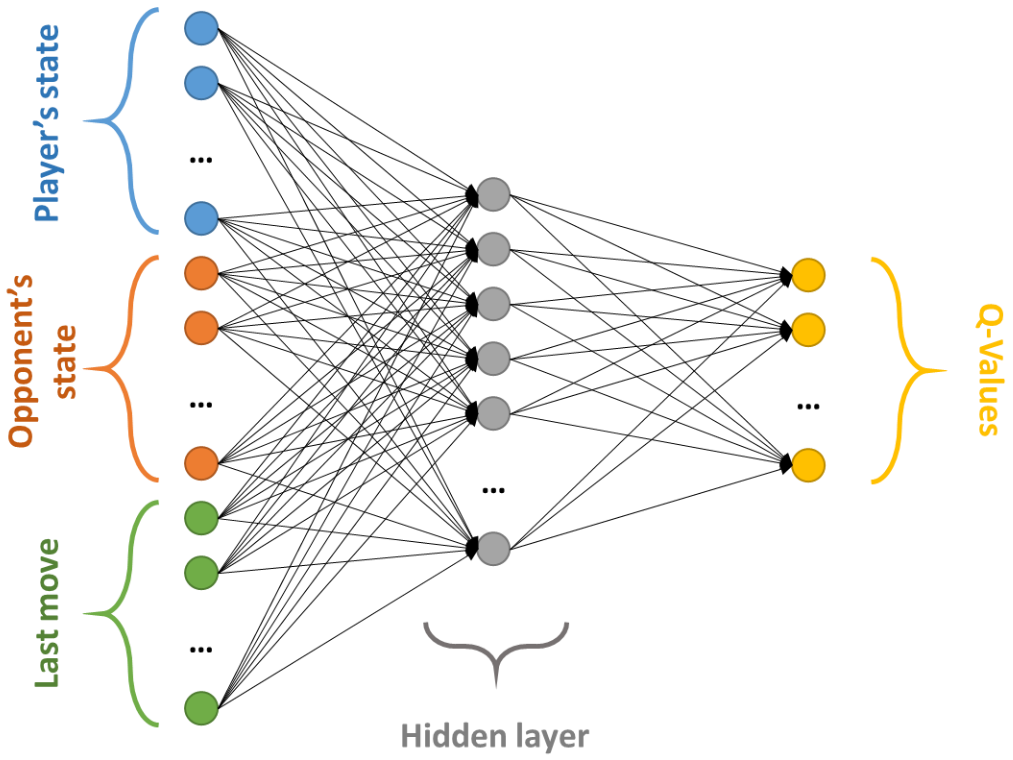


Figure 6: Neural Network Graph

Benchmark

The benchmark model for this project will be the first algorithm, which updates the learning values at the end of the game. The other two algorithms will compete against the first algorithm in a game board of 3x3 and winning line size of 3, trying to achieve less defeats in a certain number of games.

The two algorithms should outperform in terms of wins and draws than the benchmark even when the number of games is altered.

III. Methodology

Data Preprocessing

No need for data preprocessing is required for the purposes of this project.

Implementation

The implementation is organized into three categories:

- Game Engine
- Agents
- Utilities (for metrics and benchmarks)

The implementation files of this project are the following:

- **games/tictactoe_game.py:** The implementation of the game engine
- **players/abstract_tictactoe_player.py:** The abstract class that declares all the abstract methods that need to be implemented by each agent
- **players/tictactoe_computer_naive.py:** The implementation of the Tic-Tac-Toe player component using a naïve version of Q-Learning
- **players/tictactoe_computer_qlearning.py:** The implementation of the Tic-Tac-Toe player component using Q-Learning
- **players/tictactoe_computer_tensorflow.py:** The neural network implementation of the Tic-Tac-Toe player component using Q-Learning and Tensorflow Framework

- **players/tictactoe_computer_random.py:** The implementation of the Tic-Tac-Toe player component that always selects a random move. The specific component is used only for training and/or evaluation of other components.
- **players/tictactoe_human.py:** The implementation of the Tic-Tac-Toe human interface. This component receives the next move as an input from a human player.
- **run.py**: Utility for running games

During the implementation of Naïve and Q-Learning algorithms there were no specific complications, since the flow could be clearly represented in the flow diagram and the update of Q-Values is a trivial programming task. On the other hand, the implementation of Tensorflow algorithm had the complication of designing the neural network (input layer, hidden and output layer, loss function, optimizer) and implementing it using Tensorflow. The preferred approach was to keep the architecture simple (one hidden layer) and at the same time effective.

All algorithms implement both *get_next_move* and *end_of_game* methods, as declared in the abstract class. The implementation of the variations are aligned with the flow diagrams presented in the project proposal.

Naïve algorithm

The implementation of the *get_next_move* for the naïve algorithm has 3 steps. Firstly, it initializes the Q-Values for the current state and the next states based on the available seats, then it decides if the next move will be random or selected by the greedy algorithm. Lastly, it stores the result in an array, which holds the historical data.

```
def get_next_move(self, game_state):
    next_move = None
    encoded_game_state = self.__encode_state(game_state)

    self.__init_q_values(game_state)

    if random.random() < self.epsilon:
        next_move = self.__get_next_random_move(game_state)
        self.__update_epsilon()
    else:
        next_move = self.__get_next_greedy_move(game_state)

    self.game_moves_history.append((encoded_game_state, next_move))

    return next_move
```

The *end_of_game* method calculates the reward or penalty and updates the Q-Values with the result. Finally, it resets all variables that hold states during the game.

```
def end_of_game(self, winning_player_id):
    reward = self.DRAW_REWARD
    if winning_player_id == self.player_id:
        reward = self.COM_WIN_REWARD
    elif winning_player_id:
        reward = self.COM_LOSS_PENALTY
    self.__update_q_values(reward)
    self.__reset()
```

The update method updates the Q-Values with the reward or penalty for each stored set of state and action.

```
def __update_q_values(self, reward):
    for encoded_game_state, move in self.game_moves_history:
        value, times_passed = self.q_values[encoded_game_state][move]
        new_value = (value * times_passed + float(reward)) / (times_passed + 1)
        self.q_values[encoded_game_state][move] = (new_value, times_passed + 1)
```

Q-Learning algorithm

The implementation of the *get_next_move* for the Q-Learning algorithm has 3 steps. Firstly, it initializes the Q-Values for the current state and the next states based on the available seats, then it decides if the next move will be random or selected by the greedy algorithm. Lastly, it updates the Q-Values for the previous move (if exists) and stores the last move to be used on the next run of either *get_next_move* or *end_of_game*.

```
  def get_next_move(self, game_state):
    next_move = None
    self.__init_q_values(game_state)

    if random.random() < self.epsilon:
      next_move = self.__get_next_random_move(game_state)
      self.__update_epsilon()
    else:
      next_move = self.__get_next_greedy_move(game_state)

    next_game_state_score = self.__get_score(game_state, next_move)
    self.__update_q_values(next_game_state_score)

    tmp_prev_game_state = self.__apply_move_on_state(game_state, next_move)
    self.prev_game_state = self.__encode_state(tmp_prev_game_state)
    return next_move
```

The *end_of_game* method calculates the reward or penalty and updates the Q-Values with the result. Finally, it resets all variables that hold states during the game.

```
  def end_of_game(self, winning_player_id):
    reward = self.DRAW_REWARD
    if winning_player_id == self.player_id:
      reward = self.COM_WIN_REWARD
    elif winning_player_id:
      reward = self.COM_LOSS_PENALTY
    self.__update_q_values(reward)
    self.__reset()
```

The update method updates the Q-Values with the reward or penalty for the last move taken by the player.

```
  def __update_q_values(self, reward):
        if self.prev_game_state:
            learned_value = self.alpha * (reward - self.q_values[self.prev_game_state])
            self.q_values[self.prev_game_state] += learned_value
```

Tensorflow algorithm

The implementation of the *get_next_move* for the Tensorflow algorithm has 2 steps. Firstly, it finds the best move based on the outcome of the greedy algorithm on the current Q-Values and decides if the next move will be random or not. Lastly, it updates the Q-Values for the previous game state (if exists) and move and stores the last game state and move to be used on the next run of either *get_next_move* or *end_of_game*.

```
    def get_next_move(self, game_state):
        encoded_state = self.__encode_state(game_state)

        free_seats = np.where(encoded_state == self.EMPTY_SEAT)
        free_seats_t = np.transpose(free_seats)

        splitted_states = self.__split_states(encoded_state)
        q_values = self.__get_q_values(splitted_states)

        best_next_move = tuple(free_seats_t[np.argmax(q_values[free_seats])])

        if np.random.random() < self.epsilon:
            next_move = tuple(free_seats_t[np.random.randint(len(free_seats_t))])
            self.__update_epsilon()
        else:
            next_move = best_next_move

        if self.prev_game_state:
            prev_splitted_states, prev_move_t = self.prev_game_state
            learned_value = self.gamma * q_values[best_next_move]
            prev_splitted_states = [prev_splitted_states]
            prev_move_t = [prev_move_t]
            learned_value_t = [learned_value] * len(prev_splitted_states)
            self.__update_q_values(prev_splitted_states, prev_move_t, learned_value_t)

        next_move_t = np.zeros_like(encoded_state, dtype=np.float32)
        next_move_t[next_move] = 1.

        self.prev_game_state = (splitted_states, next_move_t)
        return next_move
```

The *end_of_game* method calculates the reward or penalty and updates the Q-Values with the result. Finally, it resets all variables that hold states during the game.

```
    def end_of_game(self, winning_player_id):
        reward = self.DRAW_REWARD
        if winning_player_id == self.player_id:
            reward = self.COM_WIN_REWARD
        elif winning_player_id:
            reward = self.COM_LOSS_PENALTY

        prev_splitted_states, prev_move_t = self.prev_game_state
        prev_splitted_states = [prev_splitted_states]
        prev_move_t = [prev_move_t]
        reward_t = [reward] * len(prev_splitted_states)
        self.__update_q_values(prev_splitted_states, prev_move_t, reward_t)

        self.__reset()
```

The update method updates the Q-Values with the reward or penalty for the last move taken by the player, by training the neural network.

```
    def __update_q_values(self, splitted_state_t, move_t, learned_value_t):
        self.session.run(self.q_updater,
                         feed_dict={self.splitted_states: splitted_state_t,
                                    self.move: move_t,
                                    self.learned_value: learned_value_t})
```

Utility

The utility script (run.py) implements and runs a set of games and collects a set of metrics (in combination with the time utility), such as the time of training used by the players and the outcome of the games.

The utility script, initializes the players and the game engine. The instance of the players are passed to the game engine upon initialization. It runs the number of training games defined by the user, and then a set of test games, defined by the user as well.

Refinement

The Naïve algorithm does not contain any parameters that can be refined.

Parameter *alpha* (learning rate) of the Q-Learning algorithm can be refined. However, due to the fact that the TicTacToe game is a solved game and can be perfectly played, the only reasonable value for the learning rate parameter is 1.0, which is the maximum value.

Tensorflow algorithm can be implemented with two optimizers, the Gradient Descent and the Adam Optimizer. For each optimizer, the parameter that can be refined is the learning rate. In order to identify the optimal combination of optimizer and learning rate, a number of runs have been performed. Each run executes 10000 training games between the Q-Learning and the Tensorflow and then 1000 test games. The results of all runs are depicted in Figures Figure 7 and Figure 8. The outcome of the test games proves that the optimal combination of optimizer and learning rate is AdamOptimizer and 0.001.

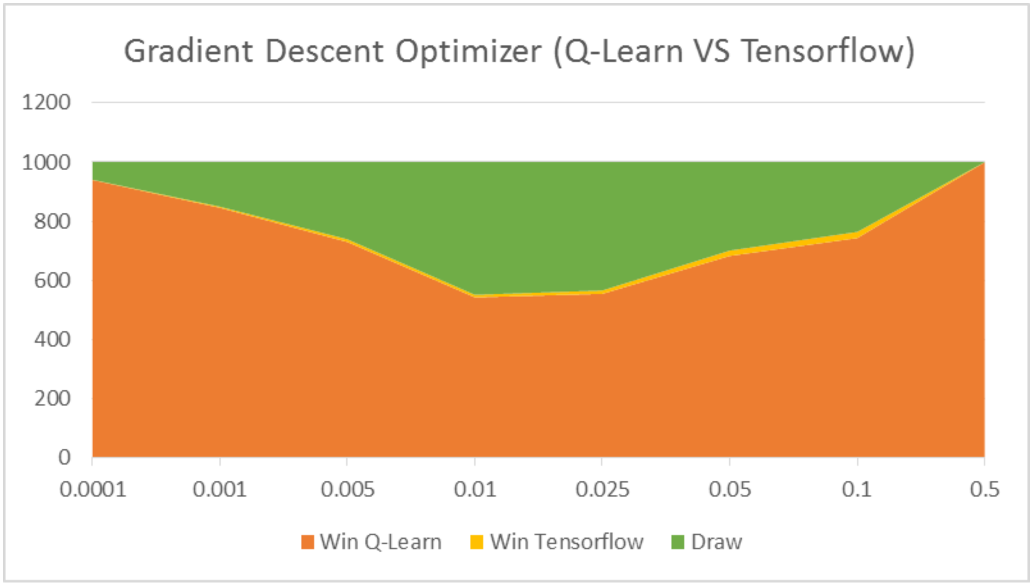


Figure 7: Gradient Descent Optimizer (Q-Learn VS Tensorflow)

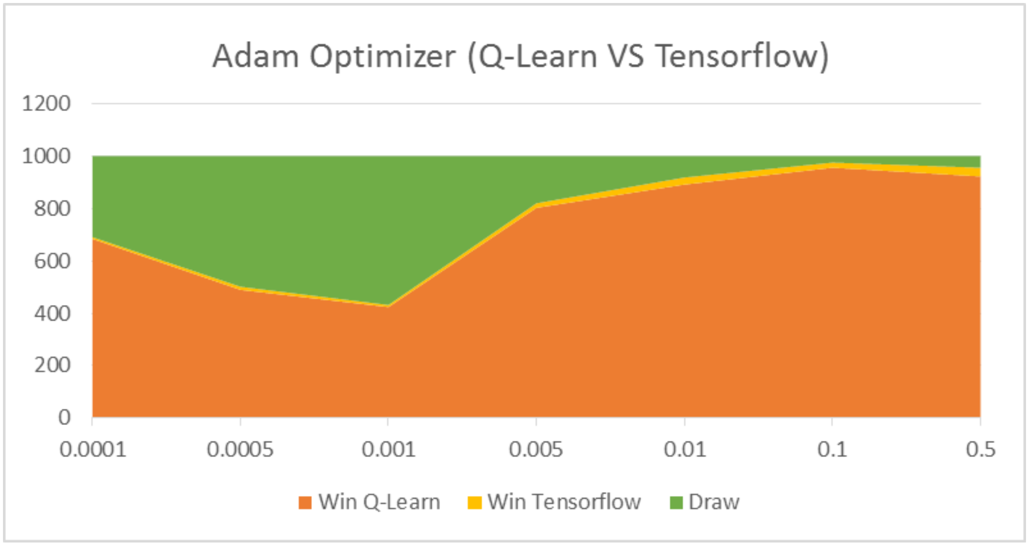


Figure 8: Adam Optimizer (Q-Learn VS Tensorflow)

IV. Results

Model Evaluation and Validation

For the evaluation of the algorithm variations of this project, a set of training games have been conducted. The number of training games ranges from 0 to 50.000 and the games have been performed between the random player and the examined algorithm variation. The metrics collected from the outcome of these games are the CPU User time in seconds (Figure 9), the CPU System time in seconds (Figure 10) and the maximum resident set size in memory in kbytes (Figure 11). The Linux utility *time* was used for the collection of these metrics.

As depicted in the diagrams, the Tensorflow algorithm requires 15 to 20 times more CPU time than the Naïve algorithm and 10 times more CPU than the Q-Learning algorithm. The Q-Learning algorithm requires 2 times more CPU than the Naïve algorithm. For all algorithms, the CPU time has a linear increase, as the number of games increases.

The maximum resident set size in memory for the Tensorflow algorithm is 7 to 8 times bigger than the Naïve and approximately 9 times bigger than the Q-Learning algorithm. The Naïve algorithm requires 1.2 times more memory than the Naïve algorithm. The memory usage remains almost stable as the number of games increases.

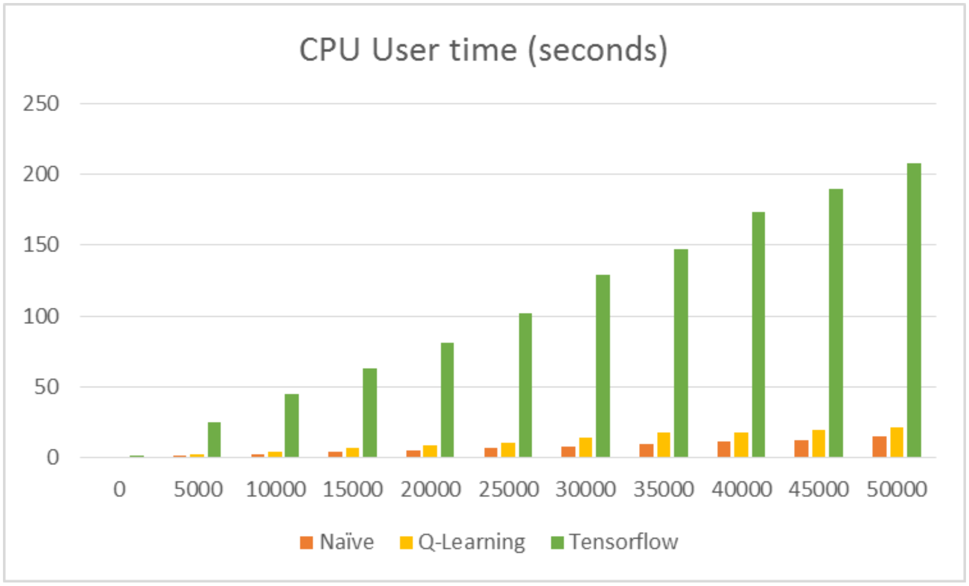


Figure 9: CPU User time (seconds)

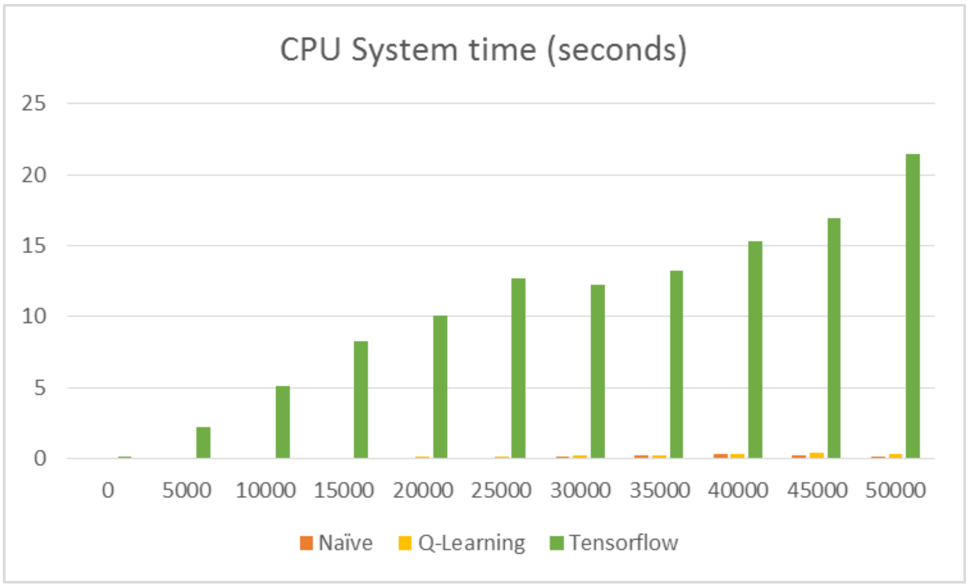


Figure 10: CPU System time (seconds)

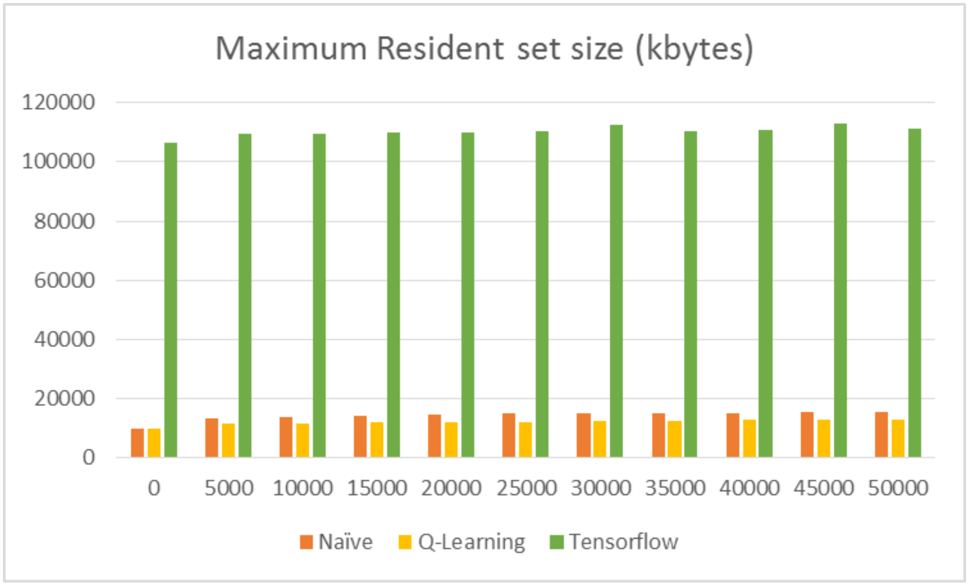


Figure 11: Maximum Resident set size (kbytes)

Justification

The benchmark for this project is the Naïve algorithm in a board of size 3x3 and winning line size of 3. The other two algorithms competed against the benchmark for various numbers of training games. After the completion of the training games, a set of 1000 test games are conducted for the evaluation of the performance.

The Q-Learning algorithm has a very high number of wins and draws against the Naïve algorithm, as depicted in Figure 12.

The Tensorflow algorithm has also a very high number of wins and draws against the Naïve algorithm, as depicted in Figure 13. The results show that in the beginning (under 10.000 training games), the wins are significantly more than the draws. This is due to the fact that the neural network has at the beginning some random Q-values, because it has not been trained adequately.

As a summary, both algorithms outperform the Naïve algorithm.

Finally, a set of training and test games are conducted between the Q-Learning algorithm and the Tensorflow algorithm (Figure 14). The results show that for training sets under 20.000, the Q-Learning algorithm outperforms the Tensorflow algorithm. After the 20.000 training sets, both algorithms perform equally (draw).

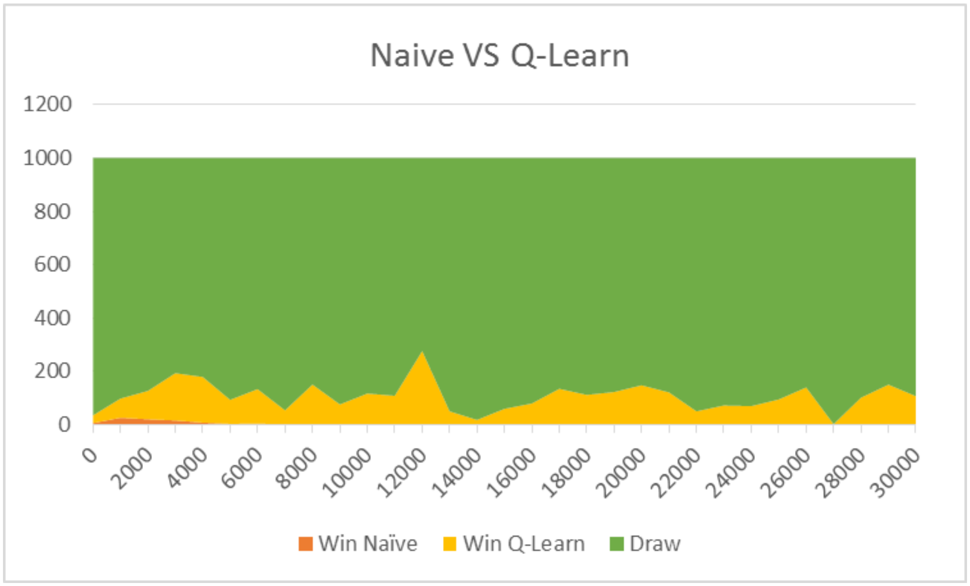


Figure 12: Naive vs Q-Learn

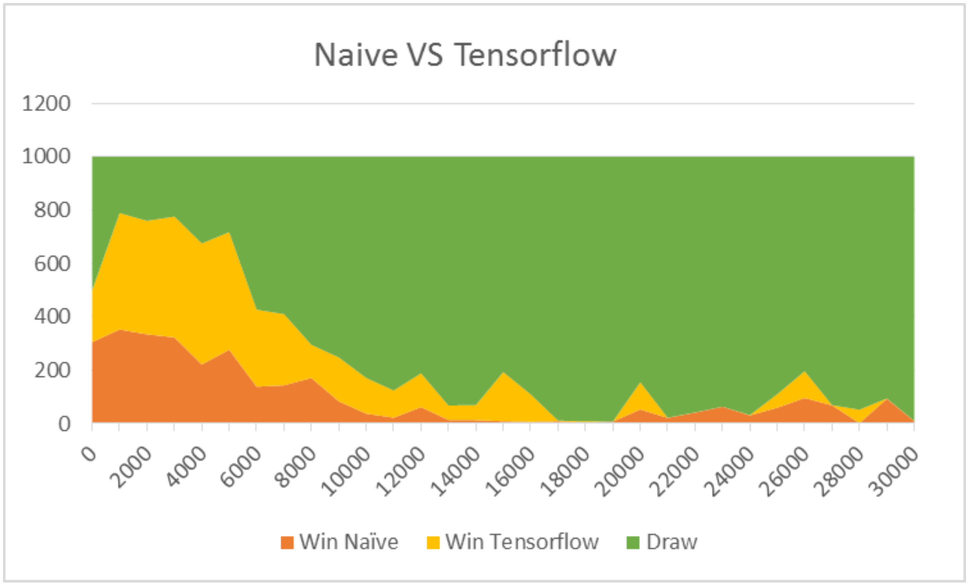


Figure 13: Naive vs Tensorflow

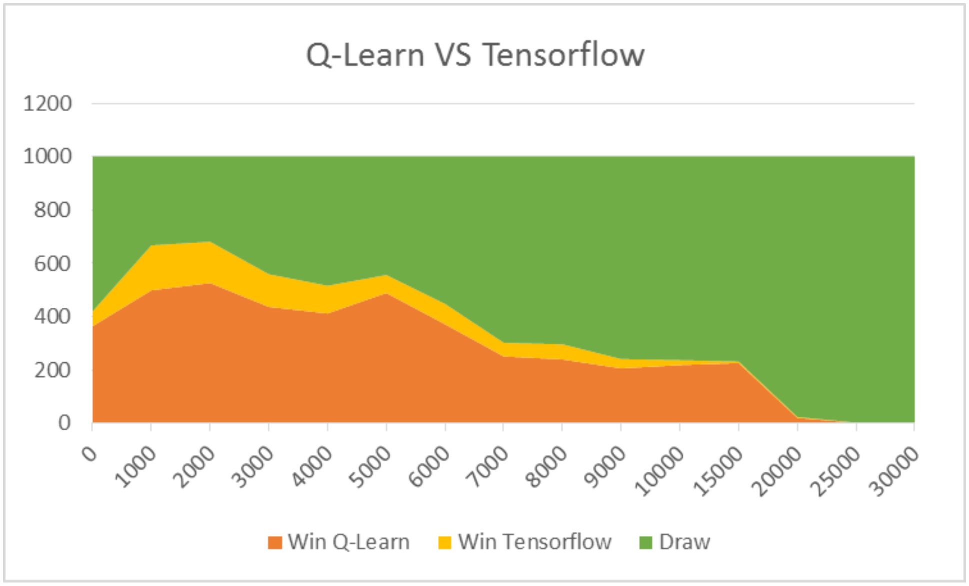
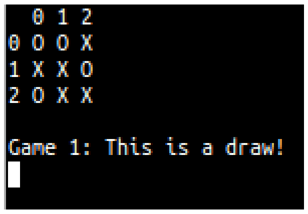


Figure 14: Q-Learn vs Tensorflow

V. Conclusion

Free-Form Visualization

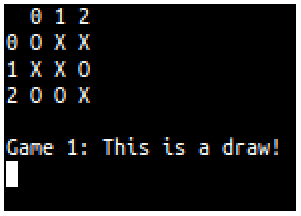
After a significant number of training sets (above 20.000), all algorithms can outperform (draw or win) a human player (Figures Figure 15 and Figure 16). Additionally, all algorithms can outperform (win or draw) Google's Tic-Tac-Toe [7] in all levels, after the algorithms are trained with a set of 20.000 or more games (Figure 17).



```
  0 1 2
0 O O X
1 X X O
2 O X X

Game 1: This is a draw!
```

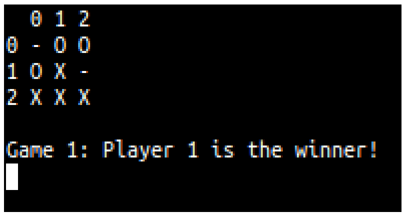
Figure 15: Human (X) vs Q-Learning (O)



```
  0 1 2
0 O X X
1 X X O
2 O O X

Game 1: This is a draw!
```

Figure 16:Q-Learning (X) vs Human(O)



```
  0 1 2
0 - O O
1 O X -
2 X X X

Game 1: Player 1 is the winner!
```

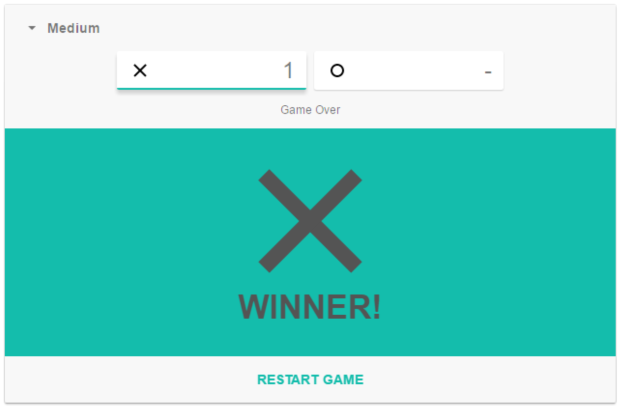


Figure 17: Q-Learning (X) vs Google's Tic-Tac-Toe (O)

[7] "Google Tic-Tac-Toe game" - https://www.google.com/search?q=tictactoe

Reflection

In this project, the goal of implementing Tic-Tac-Toe agents with Reinforcement Learning techniques has been achieved by the implementation of the three algorithms. The first challenge faced was the design and categorization of the components that consist the solution. The components are loosely coupled, thus the overall solution can be expandable, e.g. implementing a new algorithm for a player component.

The challenge in the Tensorflow algorithm was the selection of the optimizer, as well as the number of nodes in the hidden layer. The best combination was identified after a large number of performed tests.

Moreover, the game engine was implemented from scratch, without using any additional libraries, and supports Tic-Tac-Toe games with boards larger than the default 3x3 and various winning line sizes.

To conclude, all algorithms presented in this project use Reinforcement Learning techniques and are able to be trained and eventually achieve a draw or win in a game with a machine or a human player.

Improvement

The experiments prove that all three algorithms are efficient in terms of winning a game after a number of games. The training process could be accelerated by a maximum factor of 4, by applying an improvement in the Q-Values update process.

A set of state and move is provided each time the Q-Values must be updated. Due to the fact that the board is always a square, the set of state and move (Figure 18) can be rotated 90^o each time (either clockwise or anti-clockwise) and produce at most 3 more possible sets of states and moves (Figures Figure 19, Figure 20 and Figure 21). The learning value for all sets will be the same.

	0	1	2
0		**X**	
1	**O**	**X**	
2			

Figure 18: Provided board state. Next move is marked with red

	0	1	2
0		**O**	
1		**X**	**X**
2			

Figure 19: Board state and next move after 90° rotation (clockwise)

	0	1	2
0			
1		**X**	**O**
2		**X**	

Figure 20: Board state and next move after 180° rotation (clockwise)

	0	1	2
0			
1	**X**	**X**	
2		**O**	

Figure 21: Board state and next move after 270° rotation (clockwise)

Finally, another improvement would be the implementation of storing the learned values (Q-Values, Neural Network), so as to avoid executing the training phase in a later execution.