Tic-tac-toe decision maker with machine learning

Udacity Machine Learning Nanodegree Program

Final Project

Nuno Machado
21/01/2018

I – Definition

Project overview

This work explores the steps taken to build an HTML5 tic-tac-toe game. Since very young, I've been exposed to videogames, and my interest in machine-learning comes from the possibilities it opens to create artificial and challenging opponents in this kinds of games. For that reason, in my final project I chosed to implement a machine learning algorithm to try to play tic-tac-toe against a human opponent.

Tic-tac-toe is a basic game where a player must score three inline pieces in a nine pieces board. Besides being a child's game, it's not that easy to win when played with an experienced opponent. It was an objective of this project that the machine had two ways of answering to our moves, one less challenging, making random decisions, and other much more challenging, where the machine actually chooses the best move.

Problem statement

Tic-tac-toe is a deterministic, zero-sum, and perfect information game:
- Deterministic because every move drives us to a concrete state.
- Zero sum because every positive reward given to a player means a symmetric reward to the other player. The sum of both rewards must be zero.
- Perfect information, because at all times, each player knows exactly what might happen after a given move. So both players know, at any given time, all the moves available and respective consequences of those moves.

To drive a machine to make its own decisions, I implemented a minimax algorithm, where every move is pondered and evaluated. This algorithm, explained in more detail below, assumes that what is good for a player is bad for the other, trying to maximize the rewards of a player and, as a consequence, minimize the rewards of the other player. When applied correctly, this algorithm is almost impossible to beat - I use the word almost, because perhaps someone might beat it, but in my attempts I failed to do so.

To actually build a game where the minimax algorithm could be used and its results perceived, I built an HTML5 tic-tac-toe game. All game components where designed using HTML5 technologies, while the game logic and decision making was built using Python, which is faster, reliable and much easier to use in this kinds of calculations.

Metrics

The result of the algorithm should be easily perceived comparing the decisions taken by the machine and the overall results achieved on a 3x3 game when on "easy" mode, with the decisions taken on "hard" mode, after playing a big number of different games in both modes.

II – Analysis

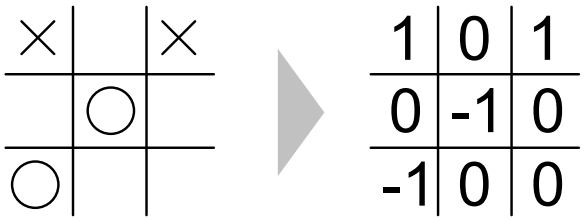
Data exploration

The only data available to the decision maker is the input produced by the player.
In a 3x3 game, on the first turn a player can choose between 9 different moves, while the next player can chose between 8 moves and so on, until someone scores a line or no moves are left.

Considering that all the pieces of the board have been moved, the total possible combinations in a 3x3 game is $2^9 = 512$, while in a 4x4 game is $2^{16} = 65\,536$ and in a 5x5 game is $2^{25} = 33\,554\,432$. Because winning boards might have some pieces still untouched, the actual number of possible game combinations are lesser than the above values for each case. Even so, it's easy to see that the amount of possibilities grows very fast with the increase of the number of rows (and columns) in the board. That huge increase can affect the performance of the algorithm when calculating all the possible moves available and respective counter-moves.

Visualization

For each tic-tac-toe game played, the board can be translated as shown in the figure bellow:



All the untouched pieces are marked as zero, while the ones moved by the player are marked as 1 and the ones moved by the machine are marked as -1. This can be abstracted using a list of lists like the following:

```
[
  [1, 0, 1],
  [0, -1, 0],
  [-1, 0, 0]
]
```

Algorithms and techniques

As stated above, the algorithm used in this game was a minimax algorithm. With this algorithm, all the decisions taken by the machine try to maximize its own rewards, while minimizing the player rewards. This technique was one of the first being used in games like tic-tac-toe, checkers or chess.

The minimax algorithm used assumes that both players choose always the best option available on each turn, with players alternating turns between moves. It also assumes that a win is better than a draw and a draw is better than a loss. Every time the algorithm has to make a decision, it gathers all the available moves, and studies the possible consequences of those moves. This can be achieved building a tree of possible moves. For each move available, the machine creates a branch for every possible counter-move (possible move made by the human player), and for each one of those, it creates a branch for each new moves available, to a certain maximum depth, where it stops (leaf). The result achieved is than evaluated using an evaluation function. In case of this game, the evaluation function studies all rows, columns and diagonals, and attributes a score every time a player is about to win - for example, in a 3x3 game, if in a given row, two pieces have a cross and the third piece is untouched, it means the crosses are about to win the game, so this row is scored. After evaluating each leaf, the algorithm chooses the best scored one. All the moves that lead to that leaf are revised in backwards, with the algorithm choosing the ones that maximize the score when it's the machine turn, and the ones that minimize the score when it's the human player's turn.

Benchmark

A good minimax algorithm should be virtually unbeatable, and with that I mean that a human player should be able, in the best cases, to draw the game, but not win it.

III - Methodology

Data preprocessing

No data preprocessing was required to complete this project. All data is analysed after each move.

Implementation

The game was made to be played in a browser, using HTML5 technologies. All the front--end does is to allow the user to select the type of game (3x3, 4x4, 5x5) and difficulty (easy, hard), make moves and to show the machine moves.

The game evaluation and decision process is built in Python, using no frameworks besides Flask, to allow simplify interaction between the browser (client) and the server.

Every time a move is made by the human player or by the machine (called CPU in the game), the browser sends the current board status to the server to check if the game is over. The game is considered over when someone wins or when there are no moves available. Because of that, the server tests the board data to check first if there is a winner and, after that, to check if there are moves available. The server then returns the results to the browser that displays them accordingly.

When it's machine turn, the browser sends a request for the server with the current board status, to allow the server to make a decision. After a decision has been made, the server responds with an object containing the row and column of the selected move.

Refinement

When considering all the possible moves and consequences of each one, the minimax algorithm can be very expensive in terms of cpu processing. While on 3x3 game that is usually not a problem, in 4x4 and especially in 5x5 games, the time the cpu takes to consider all options can be huge. As a consequence, the max_depth parameter of the minimax algorithm was reduced well below its optimal value in these cases. Because of that, the decisions taken in 4x4 or 5x5 are not always as expected. The max_depth parameter can be tweaked to achieve better results, but might turn the game unplayable.

IV - Results

Model evaluation, validation and justification

The results obtained are according to what I expected, at least for 3x3 games. For 4x4 games and 5x5 games, the max_depth of the algorithm have to be reduced to spare on the time it takes for the cpu to process the data. Because of that, the end result is not as good as expected.

Considering a 3x3 game, when on "easy" mode, where all the moves are random, it's very easy to win, and the machine doesn't have a strategy to win the game. When on "hard" mode, the machine is almost impossible to beat and, at best we can get a draw game. It's very clear that the machine has a strategy, especially when it's the machine starting the game. All the moves made by the machine are actually very smart moves, some might even surprise us. The machine performs all the calculations and measures each move, being virtually unbeatable. After dozens of games against human players, the machine never lost.

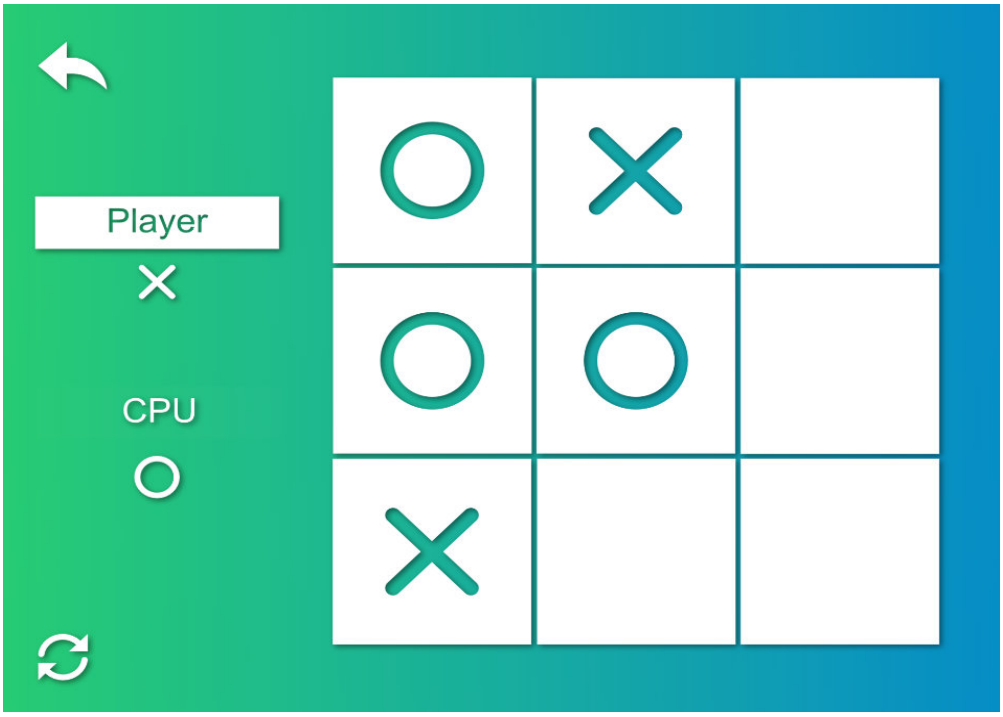
When the human starts the game, and if the player is cautious, the best the machine can do is a draw.

When the machine starts the game, it's very difficult to even draw the game. All the moves are very clever and we, as humans, in opposition to the machine, can't always make the best decision. In almost all the games, and using different strategies the machine generally wins after only 4 moves.

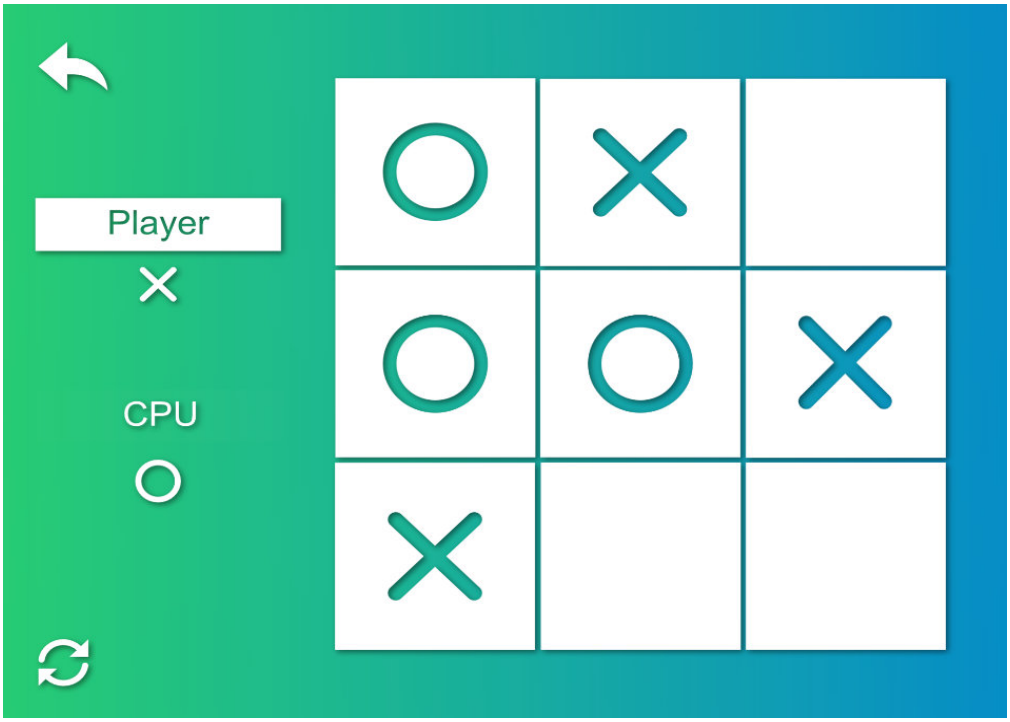
V - Conclusion

Free-form visualization

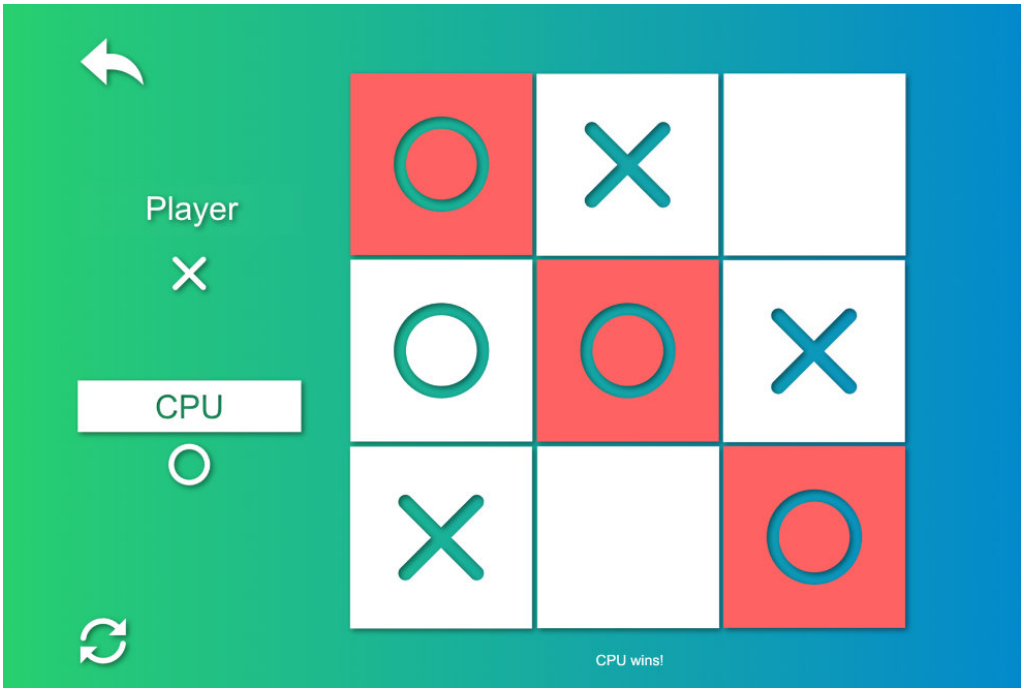
The example above proves the success of the machine in winning the game. In only four moves, and after I struggled to defend each position, the machine won.



At this point, I've already lost the game, which proves the machine has a clear strategy and the decisions it takes were correct.



Independently of my next move, the machine will win making a row or a diagonal.



And the inevitable result.

Reflection

In resume, every time the machine has to make a decision, it ponders the best move, analysing all the possible moves and consequent counter-moves, to a certain extent. The depth of the tree built by the algorithm is an important factor, and establishes a big limitation on bigger boards, where the depth was expected to be much bigger, but because of cpu processing times, has to be reduced to make the game playable.

Besides being a simple game, it's impressive the amount of math needed to make a simple decision. At first glance, the minimax algorithm might seem simple and easy to implement, but using a recursive function, the complexity it generates on execution, and the caution we might have not to put the system in an infinite loop are very impressive. Even if this algorithm is one of the most basic and simple ones in machine learning, has a great potential and I'll certainly explore it in other types of board games.

Improvement

This model has some space to improvements. A way to reduce calculation times is to store a given board and the score it gets by the evaluation function, allowing the machine to use it in future evaluations.

Another way that might improve significantly is to ignore a branch, when another branch has already a better score. That way, we can heavily reduce calculation times. The results achieved by such algorithm should be exactly the same, but with much less calculations. This can be achieved using minimax algorithm with alpha beta pruning.

Bibliography and references

"Deep Learning", Valentino Zocca *et al.*, 2017, Packt Publishing.
https://www.quora.com/Is-there-a-simple-explanation-of-a-minimax-algorithm.