Tic-Tac-Toe and machine learning

David Holmstedt
Davho304
729G43
Table of Contents

Introduction ......................................................................................................................... 1
What is tic-tac-toe ................................................................................................................. 1
Tic-tac-toe Strategies ........................................................................................................... 1
Search-Algorithms ................................................................................................................ 1
Machine learning .................................................................................................................. 2
Weights ................................................................................................................................. 2
Training data ........................................................................................................................ 3
Bringing them together .......................................................................................................... 3
Implementation .................................................................................................................... 4
Overview ............................................................................................................................... 4
UI ............................................................................................................................................. 5
TicTacToe Class .................................................................................................................... 6
Train function ......................................................................................................................... 6
Play function(s) ..................................................................................................................... 6
Board Class ............................................................................................................................ 6
Board handling ...................................................................................................................... 6
History, Successor states and Features .................................................................................. 7
Agent Class ............................................................................................................................ 7
Moves ..................................................................................................................................... 7
Weights .................................................................................................................................. 8
Critic Class .............................................................................................................................. 8
g etTraining function .......................................................................................................... 8
Discussion ............................................................................................................................... 9
References ............................................................................................................................... 10
Introduction

What is tic-tac-toe

Tic-tac-toe is a three in a row game for two people where one person plays as the “X” and one plays as the “O”. The goal of the game is to get three in a row on a 3x3 grid.

Figure 1: A tic-tac-toe game where X has three in a row.

Tic-tac-toe Strategies

The game works by presenting the players with a playing field of a 3x3 grid playing field. The players then take turns playing their respective symbol on the playing field trying to be the first one to get three in a row. The game is a so called solved game in which there is an optimal strategy that playing the game perfectly will always end up as a draw. Tic-tac-toe is different in the aspect that it is a defensive game where the first goal is to keep your opponent from getting three in a row and the second one is to try to get three in a row yourself.

Search-Algorithms

Tic-tac-toe can be solved using a Search-Algorithm, there are 362 880 different possibilities on a 3x3 grid counting invalid and game where the game should have already ended from a win. But there are in practice only 19 683 possible game situations (Hochmuth, n.d.). This is reduced even more when you take into account the symmetry of the game where the same “game” layout exist but is just rotated 90 degrees or more this in turn makes tic-tac-toe have only 827 totally unique situations and thus by just thinking about the problem reduced it from a big search problem to a relatively small one.

One of the more common solutions to the tic-tac-toe problem is using the Min-Max search algorithm which works on a basis of trying to minimize the loss and maximize the gain each step of the way down the search tree by attributing certain characteristics for the game.

Example from (Tic-Tac-Toe, 2012):

- +100 for EACH 3-in-a-line for computer.
- +10 for EACH 2-in-a-line (with a empty cell) for computer.
- +1 for EACH 1-in-a-line (with two empty cells) for computer.
Negative scores for opponent, i.e., -100, -10, -1 for EACH opponent’s 3-in-a-line, 2-in-a-line and 1-in-a-line.
0 otherwise.

Using these values the computer always tries to go for the best possible score and if implemented correctly would always end in a draw for both players.

Figure 2: A Tic-tac-toe search tree using 0, -1 and 1.

**Machine learning**

Machine learning is the use of an input and output connected to each other by a number of different weights. This is of course heavily simplified explanation of what machine learning is.

**Weights**

These weights all have different features attached to them which arbitrary values are chosen to be important for the network, these could in theory be any feature but I have decided to use the ones below.

The 6 features (MacLellan, 2013)

- $x_1 = \#$ of instances where there are 2 x's in a row with an open subsequent square.
- $x_2 = \#$ of instances where there are 2 o's in a row with an open subsequent square.
- $x_3 = \#$ of instances where there is an x in a completely open row.
- $x_4 = \#$ of instances where there is an o in a completely open row.
- $x_5 = \#$ of instances of 3 x's in a row (value of 1 signifies end game)
- $x_6 = \#$ of instances of 3 o's in a row (value of 1 signifies end game)

A row is defined as a row of 3 squares be it in rows, columns or diagonals. These features are then used to calculate the weights by reinforced learning.
Training data
The entire history of the board is used to get a list of training data, this includes the features (the six from above) in each specific history step and the current result in the form of 0 for Draw, 100 for a win and -100 for a loss. These are arbitrary numbers but is an example used by Tom M. Mitchell (Mitchell, 1997, s. 7)

Bringing them together
The Agent then uses both the History, the training data and an evaluation of that current history step (same function that is used to select a move). The evaluation is a calculation using the features and all the weights to get a single value of that move.

The calculation used:
weightX = weightX + updateConst*(currentResult - successCalc)*featureX (X is a number)

The currentResult is the training data result of that step and successCalc is the evaluated move, these two are used to update the weights towards the correct ones, the closer the estimate is to the actual result the lower the change will be and vice-versa. FeatureX and weightX is the weight number from 0-6 (7 weights) 6 uses a feature and then there is one not using the feature at all and is just a weight representing win/loss.
Implementation
The code is based on the work by Christopher J. MacLellan and will be used as a library but is modified and apparent bugs fixed.

Overview
This will be a quick overview of all the steps the program takes from the Main loop to a finished game in a bullet-point list.

- The program starts by taking a number of arguments that decides training time and if the tree will play against the player or itself.
- The Training function runs
- A loop runs through the number of training cycles that where passed as an argument.
- Agent1 (Who is the main agent) runs against another agent (Agent2) who either chooses a move totally at random or using smart moves just like Agent1
- Agent1 uses a **smartMove** where it gets the possible available moves from the board class
  - It then uses the **evalBoard** function on each available move which in turn fetches the six **features** (described earlier) and combines them into a single value and returns it
  - The returned value is then compared to every other value of the available moves and in the end selects the best move and applies it
- After that it moves over to the training part
  - The weights are set to both the critic classes (one per agent)
  - The weights are then updated using the entire board history in the Agent Class and the critic classes function **getTraining**
    - The **getTraining** function runs through the entire history by getting the features using the **getFeatures** function again but this time instead of combining it to a single value it keeps the different individual features and also checking the result +100 for win, 0 for a draw and -100 for a loss in a [[FEATURES], RESULT] return vector
    - This return is used in the **updateWeights** function located in the Agent Class
      - This runs through the entire history of the game each time using the weights and features
      - weightX = weightX + updateConst*(currentResult - successCalc)*featureX (X is a number)
- It then repeats these steps for (default 500) times
- The player can then play against the bot by using a coordinate from a number shown in the terminal
- At the end of the game the Agent will retrain from the last played set and the board resets to a new game.
UI

Figure 3: The screen after training.

Figure 4: Last game shown and information about wins when playing the user.

Figure 5: The bot playing itself.
**TicTacToe Class**

The main file for the implementation is the tic-tac-toe file where the main loops and settings for the machine learning are handled.

The available settings for the program is

1. **Number of training cycles (Default: 500)**
2. **Bot or Player (Default: bot)**
   a. This decides if the player or the computer will run against the Agent
3. **Smart Opponent (Default: False)**
   a. Use a smart opponent while training or not, default it will be random
4. **Verbose (Default: False)**
   a. If the user wants debug output

**Train function**

The Train function is where the program runs the class code to train the agents.

The function *Train* works by running the game a set amount of times (default 500) to train the Agent, the Agent can be trained against either a totally random Agent or an Agent who also uses *smartMove* to select a move. It then uses the board’s state to update the weights using training data from the *critic* class to call the *updateWeights* in the Agent class.

**Play function(s)**

The play functions are *Play* and *BotVsBot* both functions are similar in layout with small changes to allow for player input in *Play* and to use Agent2 in *BotVsBot*.

It works by starting an infinite loop through a *while true* statement the bot always plays first and after that the player can select a space to play and the loop continues forever starting a new game each time the game is done. After the game is done the Agent will update the weights using the last result and in turn getting “better” each turn.

*BotVsBot* is the same except without player input and will just run just like the Train function except it will draw it for the user and show results, but will still update the weights for both Agents after each turn.

**Board Class**

The board class is the class that handles the board for the tic-tac-toe, it is used to check the winner, handle the successor moves and get the features (hypothesis)

**Board handling**

The init function starts with creating a new board where the class creates the board to memory for later use. The structure of the board is a list with three elements which themselves have three elements in a vertical logic, the first elements is the first column.

- Example: [[0,0,0], [0,0,0], [0,0,0]]

There are functions to handle board logic, *getRows* where the functions just returns the boards rows which are in a list, similarly *getCols* returns the columns and the *getDiags*
returns the diagonals, these in turn are used to determine both the winner of the game and/or if the game is done which is done in the `isDone` function.

The next set of functions are the setting of moves, which is done in two functions `setX/setO` which sets a specific coordinate to have a 1/2 as value (which is the ID value for O and X) and saves it to history for X. And there is also a `validMove` function used to keep the player from placing in an already occupied space.

**History, Successor states and Features**

One of the main uses for the board class is the data it contains, the board contains a full history of the board states that is saved every time a move is made which means that everything is available for use. The class also has two functions called `getSuccessorsX/getSuccessorsO` which returns all the potential moves at any given time using the current board, these are then used by the Agent to choose their next moves.

Lastly there is the Features this is where the “meat” of the logic lies this is the 6 features that are extracted from every board and then used in the Agent class to update the weights later based on the Critic class training data.

**Agent Class**

The Agent class contains the actual “brain” with the other classes Board and Critic being support classes.

In the `init` function a reference to the board is set and `Weights, Player(X or O), History` is initialized and the `updateConstant` is initialized with a default value of 0.1.

The agent class contains get and set functions for the `Board` variable and the `Weight` variable as well as a `setUpdateConstant` to set how big of a change each weight update has.

**Moves**

There are two different ways the agent can make a move, one is in the `randomMove` function which just takes a Successor state at random and sets that move, and this is used to train the Agent but is not strictly necessary as you can plot two actual Agents against each other.

The other more important function is the `smartMove`, first it uses the `getSuccessorsX/O` to get all available moves it then uses the `evalBoard` function which takes the `getFeatures` return from Board class to calculate a single `bestValue`. It does this by taking the 6 feature returns and multiplies it against the weights and then add them together.

\[
w_0 + w_1*feat1 + w_2*feat2 + w_3*feat3 + w_4*feat4 + w_5*feat5 + w_6*feat6\]

It then loops through the successor states doing this calculation on all available moves to find the one with the highest value, the move with the highest value is the “best” according to the Agent and then it returns the actual Successor state.
Weights
The most important of all the functions in the Agent class is the *updateWeights* function, this is where we take the entire history of the board and loop over it.

The same *evalBoard* function from before is ran on the state and the training data from the Critic class is used and using this data the weights are updated as seen below.

\[
\begin{align*}
w_0 &= w_0 + \text{self.updateConst} \times (\text{exTrain} - \text{est}) \\
w_1 &= w_1 + \text{self.updateConst} \times (\text{exTrain} - \text{est}) \times x_1 \\
w_2 &= w_2 + \text{self.updateConst} \times (\text{exTrain} - \text{est}) \times x_2 \\
w_3 &= w_3 + \text{self.updateConst} \times (\text{exTrain} - \text{est}) \times x_3 \\
w_4 &= w_4 + \text{self.updateConst} \times (\text{exTrain} - \text{est}) \times x_4 \\
w_5 &= w_5 + \text{self.updateConst} \times (\text{exTrain} - \text{est}) \times x_5 \\
w_6 &= w_6 + \text{self.updateConst} \times (\text{exTrain} - \text{est}) \times x_6
\end{align*}
\]

Each weight has a corresponding feature.

The exTrain is the current board states “value” if the game has ended this value is either 0 for a draw, 100 for a win or -100 for a loss if the game isn’t over this value uses the *evalBoard* function to try and estimate how good the game is going using the result exTrain from the training data and the estimated result value. These have the effect of a quicker correction if the result and the estimated result has a big difference. Each cycle it updates the weights for the Agent until it has run through the entire history.

**Critic Class**
The critic class is a class to check the board history and return training data.

It has a few support functions such as *setWeights*, *setMode* which are self-explanatory and just sets values, it also has the *evalBoard* function from the Agent class.

**getTraining function**
*getTraining* is the main function of the critic class it’s a simple function in theory but is a bit finicky in the implementation, it runs through the entire history of the board and each step of the history it checks if there is a win, loss or draw and every loop it adds an entry to the *trainEx* return variable. Each item is a list [FEATURES, RESULT], the features are the ones that are returned by the board class and contains the values used to actually modify the weights and the RESULT is the result in that current step, it is 0 most of the time due to it being in a sense “draw” because there is no winner.
Discussion
The code for this type of machine learning is very simple but as a learning experience it is invaluable.

The difference between training with smart or random moves is not that different from my tests and both results in X winning most of the time if there isn’t a draw. The reason for this is probably because X gets to start each time and O hasn’t learned how to deal with this during the training. The difference between training 500 times or 50000 times has little impact on the actual end result and more research as to why this is needs to be figured out and maybe code needs to be changed.

There is ways to cheat the opponent by doing moves in one area of the grid and switching to another ultimately leading to a win because the moves are “random”. Most of the times the game will actually end up in a draw against the player, as to why this is could be due to errors in the features or not enough training. Maybe adding more features / weights could solve this problem, maybe adding a specific feature for a –

```
X   X
   
   
```

grid where two moves are apart but still in the same row, but this has to be tested of course.

Finally the system would probably be much smarter if it actually got to train against a player or a perfect search-algorithm and would be an interesting next step for the code.
References


