Decision Trees for Expert Iteration : A Benchmark Study for an Enhanced Agent Strategy

Victor Perez, *Student, University of Essex*

Abstract—Since they are born, all of them species learn trough they life from their mistakes. Our brain learns from the mistakes we make and the direct benefit they produce. Like us, intelligent systems can also learn from their mistakes by maximizing the direct reward price and a policy. In this paper I present gentle approach to Reinforcement Learning for classic games such as Oxo, Othello and Nim; using the Monte Carlo Tree Search method and Decision Trees to train two competing agents and maximize their wins. A benchmark study has been carried out upon different strategies to determine the differences between different classification methods such as Random Forests, Support Vector Machines and Neural Network. Also, the strategies test the agents performance when trained using continuous data pipeline or a 1-iteration window interval. The source code for the project is available at https://github.com/Victorpc98/CE888-Project.[6]

Index Terms—Reinforcement Learning, Decision Trees, Monte Carlo Tree Search, Machine Learning, Artificial Intelligence

1 Introduction

Reinforcement Learning (RL) is concerned with how software agents ought to take actions in a specific environment in order to maximize the cumulative or final reward. Unlike Supervised Learning, RL does not need labelled data, nor sub-optimal actions to be explicitly corrected. The focus of RL is finding a balance between exploration of uncharted territory and exploitation of the current knowledge.[1]

The goal of an RL agent is to determine a policy (a set of actions to be taken in different states of the world), so that the future reward is maximized. RL theory is based on how human brains work and how during time it learns to conduct complicated activities and decisions that couldn't been taken without a previous learning phase. Indeed, RL reassembles the learning method of human brains. It's been proved by neuroscientists that the phasic activity of dopamine neurons resembles the implementation of a prediction error from a temporal difference algorithm, in which the difference between successive predictions of future reward plus the reward available at a given time is used to learn an updated representation of the value of a given action in a particular state.[2]

The objective of these work is to train a RL algorithm that can predict the most appropriate action for an agent at every game state g_s so that the agent wins the game. The agents are trained using different strategies and training methods to maximize their winning rate. The strategies are evaluated in a deep experimentation step where 24 proposed strategies are executed and compares with each other.

- *Victor Perez is a Data Science student at University of Essex.*
- *E-mail: vp19885@essex.ac.uk*

2 Background

2.1 Monte Carlo Tree Search Method

The Monte Carlo tree search (MCTS) is a heuristic search algorithm, mainly used in game plays. It's based on the popular Monte Carlo Method, which uses randomness for deterministic problems difficult or impossible to solve using other approaches. MCTS is been proved to be the best table game tree-search improving the exponential search times of uninformed search algorithms such as e.g. breadth-first search, depth-first search or iterative deepening.[3,4]

The MCTS algorithm works simulating k random games to the very end and recording the scores. At each game step (also known as playout), the search tree is expanded based on random sampling of the search space. The final game result of each playout is then used to weight the nodes in the game tree so that better nodes are more likely to be chosen in future playouts.

As mentioned earlier, the focus of RL is finding a balance between exploration and exploitation. Exploration refers to expanding new branches that have never been explored before, while exploitation refers to exploiting the deep variants after moves with high average win rate. The formula for balancing these two aspects is known as Upper Confidence Bound applied to Trees (UCT). The Search algorithm should choose to expand the node that evaluates the bottom expression with higher value.[5]

$$UCT_x = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}},\ max(UCT_x)\ \forall\ x\ \in\ X$$

- w_i as the number of wins for the node considered after the $i-th$ move
- n_i as the number of simulations where the node was expanded after the $i-th$ move

- N_i as the total number of simulations after the $i-th$ move run by the parent node of the one considered
- c as the exploration parameter. Theoretically equal to $\sqrt{2}$
- X as the set of child nodes that could be expanded

2.2 Random Forest Classifier

Random Forests (RF) are an ensemble learning method for classification that are composed of more than one Decision Tree. The forest outputs the classification class as the average class predicted by each tree in the ensemble system. Each Tree is trained to fit the training data applying the general technique of Bootstrapping. These technique consists of generating new random samples from the given training data set. These procedure leads to better model performance because it decreases the variance of the model without increasing the bias. The idea behind it is that fitting many trees on a single training set would produce strongly correlated decision trees and so all of them would output the same prediction on the same data set. While bootstrapping the original sample to produce new similar samples is a way of de-correlating the trees by training them with slightly different training sets.[6]

3 Methodology

The RL algorithm is divided into two steps. The learning step, and the training step. These steps are repeated for a fixed number of iterations i, and for each iteration a fixed number of games k are simulated. The learning step is where the algorithm plays games to collect new data and explore new moves. For this step k games are simulated and the data is collected. Once all the games have been played, the algorithm starts the training step where it retrains itself by using the new collected data. This step can be carried out either by using the last $[i_0, ..., i_k]$ simulated iteration games, also known as continuous data pipeline (CDP) or the last simulated iteration game $i_{k_{-1}}$, also known as 1-iteration window interval (1IWI). The games are simulated by using either the outputs from a trained classifier or the outputs from a MCTS. The classifiers receive as input the game state and the agent who's playing, and it outputs the move the agent should take. At the first iteration of the algorithm, both agents are trained using a MCTS strategy in order to produce an initial data set d_t so that the classifier can be trained on existing examples. From the second iteration and on, the agents will be trained using the chosen strategy; using either the data set resulting from the last iteration d_r or the compounded data set for all previous iterations d_t

3.1 Algorithm

The initial data set d_t to train a classifier c is obtained simulating k games at the first iteration using the MCTS method. Later on, at each iteration, k games are simulated using the previously trained c and a result data set d_r is obtained from the execution. At the end of each iteration, a new c is trained using either continuous data or the output from the previous agent known as 1-window data interval.

Algorithm 1 Expert Iteration Algorithm

```
1: $continuous \leftarrow True$
2: $c \leftarrow None$
3: $d_t \leftarrow None$
4: $k \leftarrow games$
5: for $iteration = 1, 2, \ldots, i$ do
6:     if $c$ != $None$ then
7:         $d_r \leftarrow$, Simulate $k$ games using $c$ and $d_t$
8:     if $c = None$ then
9:         $d_r \leftarrow$ Simulate $k$ games using a MCTS method
10:    if $continuous$ then
11:        $d_t \leftarrow d_t$ + $d_r$
12:    else
13:        $d_t \leftarrow d_r$
14:    $c \leftarrow$ Train a new RFC with $d_t$
15:    $results \leftarrow$ Save iteration results $d_r$
```

3.2 Dataset

The system saves the board state for every movement an agent makes. It encodes the board in a one dimensional array. Later on, this data set will be used to train a classifier. Foe example, the data set produced by simulating an Oxo game consists of 11 columns. Nine columns to encode the state of each table cell. If the agent 1 occupies the cell x, then a 1 will appear at cell x. The same for the Agent 2 and if no one occupies a cell, a 0 will appear. Each round either the player 1 or 2 moves to a non-occupied cell for the given board state.

TABLE 1
Dataset for an OXO Game Simulation

Cell_0	Cell_1	$\cdots$	Cell_7	Cell_8	Move	Agent
1	0	$\cdots$	0	2	2	1
1	0	$\cdots$	0	2	6	2
1	0	$\cdots$	0	2	1	1
1	1	$\cdots$	0	2	5	2
1	1	$\cdots$	0	2	7	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

The same thing applies for the Othello game. As in the Othello game the board is a 2 dimensional space, it's transformed into a 1 dimensional array.

TABLE 2
Data set for an Othello Game Simulation

Cell_0_0	Cell_0_1	$\cdots$	Cell_3_2	Cell_3_3	Move	Agent
0	0	$\cdots$	0	1	03	2
0	0	$\cdots$	0	1	01	1
0	0	$\cdots$	0	0	20	1
0	0	$\cdots$	0	0	10	2
0	0	$\cdots$	0	0	00	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Finally, for the Nim game, as the board state is composed of the remaining chips, we only have to encode 3 variables, the remaining chips, the move that the agent took and the agent itself.

TABLE 3
Data set for a Nim Game Simulation

Chips	Move	Agent
15	3	1
12	3	2
9	3	1
6	3	2
3	3	1

4 Results

In this section are presented the results obtained after executing the program for 20 iterations and 10 games per iteration for the Oxo, Othello and Nim games using different strategies.

4.1 Competing MCTS Agents

Two competing agents that follow a MCTS strategy are a clear example of an strategy that does not improve over time and neither of them outperforms the other agent. The agent won't be able to improve their moves over time because they're using the MCTS search algorithm to find the best move at the present state but without evaluating past moves. Therefore, each game is played without knowledge of past games.

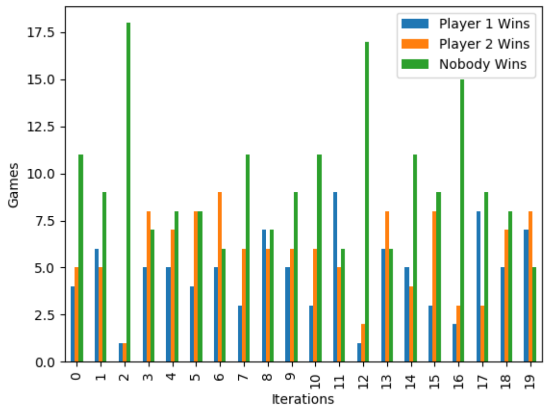

Fig. 1. Competing MCTS Agents for $i = 20$ and $g = 20$

4.2 Competing RFC Agents

Now, a different strategy is used. For every iteration, a RF is trained using past data from the agent. Therefore, the agent is learning to play based on decisions he made on earlier games. This can be seen as an evolutionary agent, where for every iteration a new generation is born, ideally being better that the previous generation

At Figure 2 we can observe that both agents start to compete with each other and improve their moves over time. We can observe that the agent started to play defensive (having a high rate of overall tails) and quickly change it's strategy and started playing more aggressively (increasing it's win rate).

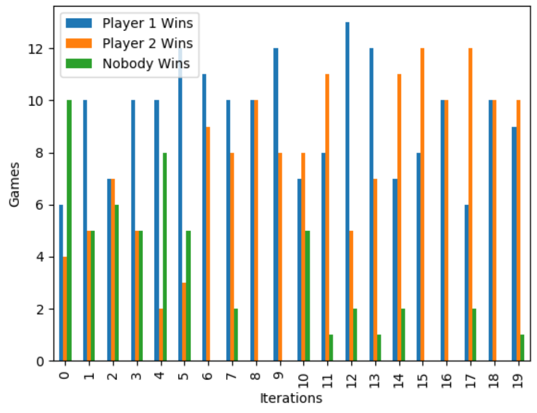

Fig. 2. Competing RFC Agents for $i = 20$ and $g = 20$

4.3 RFC Agent VS MCTS Agent

Since both agents are improving at the same time and competing with each other we cannot see any clear improvement of one agent above another. What if the moves of the Agent 1 are determined using a RFC and the moves of the Agent 2 with a MCTS?

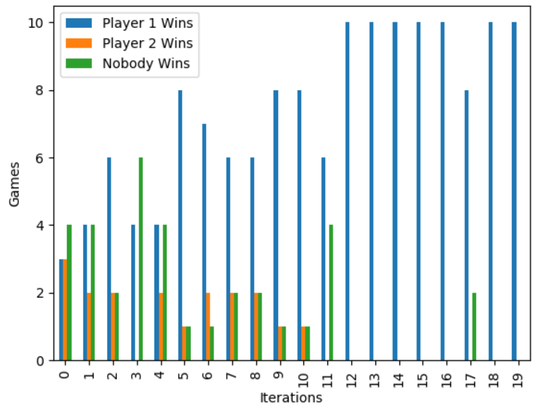

Fig. 3. Competing RFC and MCTS Agents for $i = 20$ and $g = 20$

The results show that the Agent 1 is improving a lot. Every next iteration it's improving its performance and the Agent 2 is performing worst.

4.4 Benchmark

In this section a deep experimentation process is conduced to find the best strategy to follow when playing Oxo, Othello or Nim. In order to conduct a fair experimentation and evaluation the following metrics has been used.

$$w_{a^n} = \frac{a_w^n}{g_n}, \text{ where } w_{a^n} \in [0,1] \tag{1}$$

$$a_{score}^n = \frac{w_{a^1}}{w_{a^2}} \tag{2}$$

- g_n as the number of games played
- a_w^n as the number of wins for agent n
- $cont$ determines if the strategy uses a continuous data pipeline
- w_{a^n} as the win rate for an agent n
- a_{score}^n as the game score for agent n
- t as the tails rate
- a_s^n as the strategy followed by agent n

Where an a_{score}^n close to 0 means that the strategy the agent is following isn't performing well, whereas a value above 1 means that the strategy the agent is playing is performing better than the other agent. As high gets the a_{score}^n, the better the strategy works. The score is useful to determine the overall performance of a strategy not only taking in account the agent win rate but also the opponent win rate and the tails rate. Two different strategies where both of them have the same w_{a_1} but different w_{a_2} could seem to be performing equally, but the one where w_{a_2} is higher performs worst since the a_2 increases its winnings and the game reduces its overall tails.

The following table comprises the results for an extensive experimentation where 24 strategies have being tested. For each strategy, the algorithm runs 20 iterations and 10 games per iteration, simulating 200 games in total. Also, some strategies apply the so-called continuous training using a continuous data pipeline, whereas some of them, only use the last iteration game results to fit a classifier, also known as the 1-iteration window interval.

The results obtained are promising. Although it differ from game to game, we can see that the RF normally outperform other strategies. For the Oxo game, the best strategy is the #4 with a score of 93.5. It doubles in score the second best strategy #3. The main difference between #3 and #4 is that the strategy #3 is using a continuous data pipeline to train the RF. We clearly observe that using a 1-iteration window performs much better than using the accumulated results. This suggests that it's not a good idea to train the classifier using early versions of the generation but with the results form the latest generations. As the agent is learning on time and becomes more intelligent as new generations are trained, using it's early generation results as input for a new generations makes it perform worse since the early versions of the agent are worse than older generations.

Similar results are obtained for the Othello game, where this time the strategy #11 following a RF with a 1-iteration window interval performs better than the same strategy but using a continuous data pipeline #12. Nevertheless, the RFs aren't the best performing strategy but the SVM at the strategy #14 with a score of 14.

On the other hand, for the Nim game the strategies that best perform are the #20 and #21. It's surprising to see how a RF strategy with a continuous data pipeline #19 performs really bad when compared to other strategies and in special with its sister strategy #20. Unlike at the Oxo game, where the RF strategy following a 1-iteration interval outperformed the continuous data pipeline doubling it's score, now it's the other way around. The 1-iteration interval is being outperformed by 16 times by the continuous data pipeline strategy. Discordant to what was exposed for the Othello game, for the Nim game using a continuous data pipeline results not only in outperforming the 1 window interval strategy but also all of them strategies proposed. It's therefore the early game results that lead the agent to outperform other strategies.

TABLE 4
Strategies Benchmark

#	g	a_s^1	a_s^2	$cont.$	w_{a^1}	w_{a^2}	t	a_{score}^1
1	oxo	mcts	mcts	yes	0.24	0.215	0.545	1.11
2	oxo	mcts	mcts	no	0.235	0.225	0.54	1.05
3	oxo	rfc	mcts	yes	0.92	0.02	0.06	46
4	**oxo**	**rfc**	**mcts**	**no**	**0.935**	**0.01**	**0.055**	**93.5**
5	oxo	svm	mcts	yes	0.905	0.02	0.075	45.25
6	oxo	svm	mcts	no	0.84	0.015	0.145	56
7	oxo	mlp	mcts	yes	0.88	0.025	0.095	35.2
8	oxo	mlp	mcts	no	0.92	0.025	0.055	36.8
9	othello	mcts	mcts	yes	0.52	0.48	0	1.08
10	othello	mcts	mcts	no	0.49	0.51	0	0.96
11	othello	rfc	mct	yes	0.85	0.09	0.06	9.44
12	othello	rfc	mcts	no	0.815	0.13	0.055	6.26
13	othello	svm	mcts	yes	0.87	0.11	0.02	8
14	**othello**	**svm**	**mcts**	**no**	**0.91**	**0.065**	**0.025**	**14**
15	othello	mlp	mcts	yes	0.825	0.11	0.065	7.5
16	othello	mlp	mcts	no	0.885	0.095	0.02	9.3
17	nim	mcts	mcts	yes	0.495	0.505	0	0.98
18	nim	mcts	mcts	no	0.55	0.45	0	1.2
19	nim	rfc	mcts	yes	0.665	0.335	0	2
20	**nim**	**rfc**	**mcts**	**no**	**0.975**	**0.025**	**0**	**39**
21	nim	svm	mcts	yes	0.97	0.03	0	32
22	nim	svm	mcts	no	0.945	0.055	0	17.2
23	nim	mlp	mcts	yes	0.945	0.055	0	17.2
24	nim	mlp	mcts	no	0.91	0.09	0	10.1

5 Discussion

5.1 Future Work

My future work consists of going deeply in the experimentation process to understand better how agents are improving and which are the key parameters that best perform, as well as understanding why in some games such as Oxo using a RF strategy with a 1-interval window is highly effective against a RF continuous data pipeline, whereas for the Nim game it's the other way arround.

- Understand how the complexity of the game requires different techniques and strategies.
- Parameter tuning for all the classifiers.
- Experimentation with bigger window intervals.
- Introducing more sophisticated games such as Go.
- Introducing new types of Neural Networks such as a Generative Adversarial Network (GAN) or a Recurrent Neural Network (RNN).
- Training both agents with different strategies.
- Introducing the effect on performance when the agent starts the game and otherwise.

6 Conclusion

The results obtained are good. On one hand, we can see that when both agents compete with each other and they are using a RFC to determine their moves, they improve in such similar ways and they choose to play aggressively. Therefore, the number of ties decreases to almost none, while the wins stay balanced between agents.

On the other hand, when the moves of an agent are determined using a classifier, it progressively starts increasing its winning rate and perform much better than the other agent that's using MCTS to determine its moves. This happens because the agent moves are being determined based on the outcomes of past generation product of a learning process that enables the agent to perform better at each generation. The MCTS agent is not learning from past games and is unable to improve it's movements or identify bad movements that in other games lead him to loose.

Random Forests are a strategy that works with the three proposed games. It's the strategy that in most cases works the best obtaining the higher scores. Nevertheless, as exposed at the experimentation, training the classifier with a continuous data pipeline or a 1-iteration window directly affects the strategy score. Easy games such as Nim seem to obtain much better scores when using a 1-iteration window interval that a continuous data pipeline. On the other hand, much difficult games where there exist much more possibilities at each time step, it seems that following a 1-window interval strategy still outperforms following a continuous data pipeline being the score differences between each other slightly smaller.

On overall, training an Oxo, Nim or Othello agent using Decision Trees has resulted in a strong agent that in few iterations is able to outperform other strategies in most cases.

References

[1] Richard S. Sutton and Andrew G. Bar, *Reinforcement Learning : An Introduction*, 1rd ed. MIT Press, Cambridge, MA, 2015.

[2] J. P. O'Doherty, S.W. Lee, D. McNamee, *The structure of reinforcement-learning mechanisms in the human brain*, 1rd ed. California Institute of Technology, 2014.

[3] Sagar Sharma, *MCTS For Every Data Science Enthusiast*, 1rd ed. Aug 1, 2018

[4] Magnuson, Max, *Monte Carlo Tree Search and Its Applications*, Vol. 2: Iss. 2, Article 4. 1rd ed. University of Minnesota, 2015.

[5] Konstantia Xeno, Georgios Chalkiadakis, Stergos Afantenos *Deep Reinforcement Learning in Strategic Board Game Environments* University of Crete, February 2017

[6] Breiman, L, *Machine Learning*, Volume 45, Issue 1, pp 5–32. October 2001

[7] V. Perez, *Decision Trees for Expert Iteration*, GitHub repository, https://github.com/Victorpc98/CE888-Project. University of Essex, February 2020